②

RL-TR-92-343
Interim Report
December 1992

PERFORMANCE MEASURES OF PARALLEL DIGITAL SIGNAL PROCESSOR SYSTEMS

The MITRE Corporation

Joel D. Harris

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-92-343 has been reviewed and is approved for publication.

APPROVED: Paul M. Engelhart

PAUL M. ENGELHART
Project Engineer

FOR THE COMMANDER:

JOHN A. GRANIERO
Technical Director
Command, Control and Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL(C3CB)Griffiss AFB NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)

2. REPORT DATE
December 1992

3. REPORT TYPE AND DATES COVERED
Interim Sep 91 - Aug 92

4. TITLE AND SUBTITLE
PERFORMANCE MEASURES OF PARALLEL DIGITAL SIGNAL PROCESSOR SYSTEMS

5. FUNDING NUMBERS
C - F19628-89-C-0001
PR - MOIE
TA - 71
WU - 50

6. AUTHOR(S)
Joel D. Harris

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The MITRE Corporation
Burlington Road
Bedford MA 01730-0208

8. PERFORMING ORGANIZATION REPORT NUMBER
MTR 11263

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3CB)
525 Brooks Road
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-92-343

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Paul M. Engelhart/C3CB/(315)330-4063.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report summarizes a portion of the research accomplished under MITRE MOIE 7150, entitled "Parallel Signal Processing." The work performed for this specific task concentrated on developing and presenting useful measures of performance for real-time parallel digital signal processor systems. Using the Common Signal Processor (CSP) as a testbed for the investigation, a three level method of measurement was followed. The results were used to evaluate the strengths and weaknesses of the CSP in processing selected applications and to provide insight into its general capability. A discussion of issues related to the process of benchmarking in general and lessons learned is also included.

14. SUBJECT TERMS
Parallel Processing, Signal Processing, Performance Measures

15. NUMBER OF PAGES
64

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
UL

ABSTRACT

Methods developed to measure the performance of conventional computer systems have limited applicability to the class of computers that have been designed recently to execute real-time digital signal processing tasks. The work described here addresses the question of how to develop and present useful measures of performance for these types of computers. Using the Common Signal Processor (from IBM) as a testbed for the investigation, a three level method of measurement was followed. One level focuses on the architecture of the CSP, characterizing the physical limits of the system components and the process of software development. The other two levels focus on operational characteristics, the first level on abstract tasks running at maximum speed and the second on tasks implementing real applications. The results are used to evaluate the CSP's strengths and weaknesses in processing selected applications and to provide some insight into its general capability. A discussion of issues related to the process of benchmarking in general and lessons learned from work performed on the Common Signal Processor is included.

TABLE OF CONTENTS

SECTION PAGE

LIST OF FIGURES

FIGURE PAGE

LIST OF TABLES

TABLE PAGE

SECTION 1

INTRODUCTION

Many advanced signal processing systems require very high speed computation, multiple modes of operation, and high reliability, all within strict size, weight, and power constraints. Demands for future applications that seek to improve detection of low signal-to-noise ratio targets and to improve resistivity to jamming or interference continue to drive the search for greater performance within even tighter constraints. Examples of these applications are avionics systems, anti-submarine warfare systems, and integrated radar and electronic warfare suites. As these systems grow in complexity and their procurement becomes increasingly costly, programmable processors that can be used in a wide range of systems are being viewed as a way to reduce nonrecurring engineering costs and ensure the flexibility needed to meet evolving requirements. However, they represent a departure from traditional signal processor design, in which the tendency has been to build application-specific hardware for each individual system. The challenge in the design and use of a programmable parallel processor is to optimize performance to a level comparable with that previously attained only by application-specific hardware.

Recent advances in parallel processing technology have made it increasingly practical to build programmable parallel signal processors that meet the performance and physical requirements of embedded signal processing systems. In fact, such processors have been built and are being introduced in current procurements. Details of their architectures vary, and the skills and tools needed to program these processors are highly specialized. The objective of the Parallel Signal Processing project is to develop general techniques that make it easier to understand, evaluate, and design programmable parallel signal processor architectures. These techniques would be used to provide insight on current procurement decisions as well as to develop next-generation programmable parallel signal processor design concepts. The project consists of four tasks: 1) parallelism analysis of signal processing algorithms, 2) development of formal methods for mapping algorithms onto parallel signal processors, 3) performance measurement of parallel signal processor systems, and 4) system-level performance modeling of parallel signal processors. This report describes the results of the first year's activities on the performance measurement task. The other three activities are documented in separate reports [1,2,3].

Techniques for measuring the capability of a computer's performance have long been important to the development and use of computers. Such measurements give the user information about how well a particular machine can perform a class of problems. They can also be used by the system developer to identify a particular weakness or strength in the system, or to quantify the contribution that a particular aspect of the system architecture makes to the system performance. Measurement techniques have come to be called benchmarks because the most commonly used measurement method has been to apply a well defined and commonly accepted task or task set to a system and to measure the time the system takes to complete it. This provides a "benchmark" by which to compare one system against another.

Methods developed to measure the performance of conventional computer systems have limited applicability to the class of computers that have been developed recently to perform real-time digital signal processing tasks. Benchmarks such as Whetstones or Dhrystones may reveal how fast one processing element of the system can perform floating point arithmetic operations, but it sheds little light on more important issues for these systems. Those issues include: 1) whether the system can meet the real-time constraints of a given application running in continuous operation, 2) to what degree is parallelism in the system providing speed-up to the problem computation, and 3) whether the system data transfer resources are adequate to handle the requirements of the application and, if so, are those resources efficiently utilized.

The uniprocessor benchmarks are inadequate because the standard computation model by which single processor systems are understood and measured does not fully describe a multi-processor system, where more than one thread of instructions is being executed at a time. The ability of a single measure to predict an application performance depends on how the application is executed on the system. A generally applied multiprocessor programming model is needed to find measures that will predict how an application will perform on different architectures.

In a thoughtful paper on benchmarking parallel processor systems, G. Lyon explains that in the absence of such a model, there is a spectrum of approaches to system measurement [4]. There is on one end the application of a large representative program to the system. This approach, if the program is successfully applied to the system, can provide information on the structural strengths of the system in relation to the application being used. It may fail, however, to pinpoint system weaknesses (that may pertain to the function of a different algorithm). It is also very time consuming to apply a complete program to each system under measure due to the absence of a high level language common to any two systems. On the other end of the spectrum, specific metrics may be derived by exercising individual elements of the system, such as, the speed of multiply and accumulate operations on one processing element. This approach provides clear information on the weaknesses of specific elements of the system, but there is currently no method to extrapolate the measures to predict high end system performance.

A general goal of system measurement is to find program modules with the highest level of generality to compare against other existing systems and applications. The Parallel Signal Processing project seeks, as a long term goal, to improve generally applicable programming models to help accomplish this. A method is used in the work described here that combines aspects of both ends of the spectrum that G. Lyon presents. It is considered a first step in the process of seeking an improved programming model to be used for benchmarking parallel signal processing systems.

In the current state-of-the-art, the question of how to develop and present a useful set of measures to describe high performance systems remains open. The viewpoint taken in this report is based on the premise that current methods of performance analysis require substantial hands-on experience on a system to be able to characterize it to a useful degree. Our work, therefore, began with choosing a system to which we would have free access. The work described here applies system measurements to our first target system, using some typical radar processing algorithms for measuring performance.

The Commc . Signal Processor (CSP) was chosen as the first system on which to begin our work. It i. a high performance modular multiprocessor Digital Signal Processor system designed and built by IBM under contract with the U. S. Air Force. The system meets the needs of our work in a number of ways. The most important is that it is typical of the class of reai-time programmable systems that we are investigating, as described in section 2.3. Furthermore, at the time we began, the system included a complete usable set of both hardware and software. The software includes a full detail system level behavioral simulator, by which we could conduct our tests over an extended period at MITRE. This gave us the time to learn the system programming language, develop measurement applications, apply them to the system, and analyze results in an iterative process. We were also encouraged and assisted by the extensive cooperation from IBM through the loan of necessary software modules and technical support. This paper presents the results of our measurement of the performance of the IBM CSP.

The work is described in four sections. The first section describes measures of interest and provides definitions and a framework within which to view system measures. The next section describes the architecture of the CSP at a high level and the work that was performed on the CSP to measure system performance. This includes the system resources and facilities required to do the work and a description of the process of software development. The next section contains results of the system measurements, along with the method used for obtaining the results. A number of details of the system operation are described and system performance numbers are given. These results are then used as the basis for a general evaluation of the system, with a discussion of its strengths and weaknesses in processing selected applications. The conclusion contains a discussion of issues related to the process of benchmarking in general and lessons learned from work performed on the Common Signal Processor.

SECTION 2

DEFINITION OF PERFORMANCE MEASURES

2.1 PURPOSE OF PERFORMANCE MEASURES

Performance measures cover many aspects of the operation of a machine and vary from subjective user preferences to objective clock cycle measures. They must be viewed in the context of a purpose for developing the measures. Our motivations for quantifying measures of performance for these systems include:

1. To choose a processor for performing a set of tasks or for use in a particular system and judge a system's spare capacity for handling future tasks.

2. To tune the configuration of a processor system or the mapping of an application onto that system to best perform a given application.

3. To examine the strengths and weaknesses of an architecture for the purposes of improving future architecture designs.

4. To identify measures and methods of measuring that improve the process of benchmarking by increasing the efficiency of the process and by adding insights to the system under test and to DSP systems in general.

2.2 MEASUREMENT CONCEPTS

The exact definition of a performance metric often depends on the context. In this section, we present measurement concepts and define terminology that will be used in this report. The generic data transfer network shown in figure 1 is used to illustrate exact measurement points; however, the metrics may be used for all system components. Some of these metric definitions are fairly standard, such as throughput and latency [6,7]. Others are less commonly used, and may in fact be called by different names elsewhere.

Theoretical peak measure - This measure is the one most often cited in system marketing literature and is derived from the physical characteristics of the system, such as clock cycle time, bus width, buffer size, and processor instruction execution rate. It does not take into account such constraints as context switching time, configuration change overhead or buffer capacity limitations. It is used to describe the physical parameters of the system and is a basis for the measure of efficiency. For example, peak throughput in figure 1 is derived from the product of the clock speed and the data path width.

Abstract measure - This refers to the best case performance, within the context of the operating system, but not in the context of an application. This means that control function overhead is taken into account, but utilization and mapping constraints are not regarded. In our example, for instance, the continuous rate of data transfer is measured with the

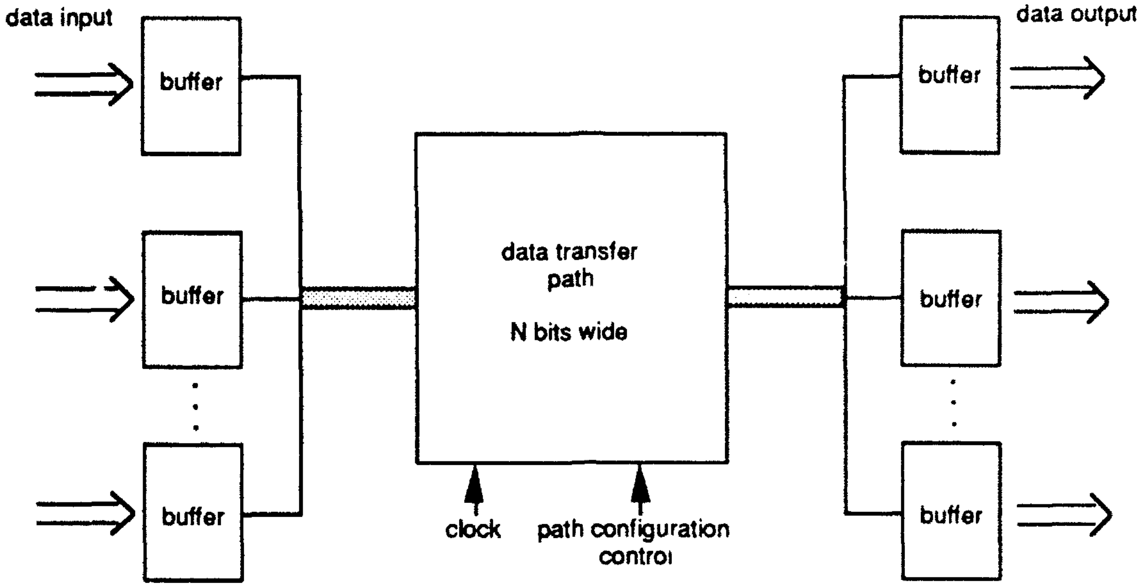


Figure 1. Generic Data Transfer Network Example

assumption that the data is constantly available at the input, there are no buffer overflows or data path conflicts, and the data is constantly accepted at the output.

The abstract measures can be compared to the peak measure for insight into system efficiency and to the application measure for insight into utilization levels, as defined below.

Application based measure - This is the measure of the performance of a resource or set of resources as the system is performing an actual application. This measure is correlated to a particular application and provides information about the match of the application to the system architecture, especially when it is compared to the abstract measures. This would apply to our example as an application specified task with a fixed number of inputs, block lengths and input rates, under the control of system program mechanisms.

Application computation cycle (or processing cycle) - This cycle refers to the repetitive nature of DSP operations, and is defined as the time necessary to complete the processing on a data block or vector that enters a functional element, or the system viewed as a whole. For one element programmed to perform one subset of operations on the data, the computation cycle is the time for it to complete a full set of computations that are repeated continuously. In our example, the application cycle could be determined by the arrival of data from one radar system dwell or one picture frame of a digital camera.

Utilization (% time in use) - The aggregate time a resource or set of resources is being put to use during the cycle (i.e., not available to perform another task) divided by the total time of the cycle, expressed as a percentage. This metric is useful particularly for measuring the effectiveness of resource allocation for an application and for judging the need to add more resources to the system to gain higher performance. The utilization measure applies in oui example to the buffers as the average of space filled with incoming data and to the transfer path as the percentage time the data transfer path is occupied with transferring data.

Throughput - This measure may apply to a single system element, group of elements, or the system as a whole. It is defined as the amount of data to move through or to be processed per unit time. Throughput is most commonly used to describe computation speed. This definition includes other types of data movement as well, reflecting the concern in digital signal processing applications for the sustained movement of data through the system. For instance, in our example, the throughput is the sustained rate at which data arrives at the combined outputs.

Operating speed - The throughput while the device is in use. This can be derived by dividing the throughput by the utilization. In our example, for instance, this is the rate at which data is transferred over the data path only while the path is actually operating and does not include time that it is waiting for data to arrive at the transfer point or setting up the transfer path.

Total efficiency - The throughput, divided by the theoretical peak throughput, expressed as a percentage. This measure reflects how well the application structure matches the system architecture or the ability of the system to fully utilize the system components. In our example, efficiency reflects the percentage of time that the data transfer path is in full operation performing a data transfer.

Operating efficiency - The operating speed divided by the peak throughput, expressed as a percentage. This measure is useful for judging the operation of a particular system element, rather than overall system performance. A comparison of operating efficiency with total efficiency can be used to identify performance bottlenecks or potential for the system to accommodate larger applications.

The distinction between total efficiency and operating efficiency may be important and subtle. For instance, data arriving at a data transfer network at a rate much slower than the transfer bandwidth of the network must be buffered at the input if the full bandwidth of the transfer path is to be used. If the buffer space is not adequate, the operating efficiency of the transfer path is degraded. On the other hand, if buffer space is adequate, the data may be transferred in bursts, leading to higher operating efficiency, but not higher total efficiency. Differences in the two measures may identify poor system utilization or spare system capacity.

Latency - This measure applies to a single system element or group of elements and is defined as the amount of time between the moment the first datum in a block enters the element and the moment the first datum from that block comes out of the element. In the data transfer example, latency is the time between the arrival of a word at the input and arrival of the same word at the output. This is to be distinguished from throughput, which is the rate at which the words arrive at the output. Additional buffer capacity may help increase the network throughput by allowing the transfer path to operate more at full speed, but with an increase in the latency.

2.3 TARGET CLASS OF PROCESSORS

Our measures are focused on a particular class of machines. The target class is defined as real-time parallel digital signal processor systems and is characterized and restricted by the following aspects:

1. Programmability

Software to implement signal processing functions and system control is generally written on more than one level, usually distributed between data flow and computation, and sometimes computation is distributed between high-level code and macro code. It is possible that this will be less so as software development tools become more sophisticated, but is currently an important characteristic with respect to performance measures.

2. Real-time operation

During operation, results are calculated on data that arrives as a continuous stream and flows through in sections (on blocks of data) that are independent of each other. Operations are generally repetitive for each of the blocks and final outputs are not stored.

3. Dedicated processing

The processing being performed at a given time is for a single application, i.e., there is no time sharing between applications. This restriction does not exclude the performance of multiple tasks in one application.

4. Modular

The architecture is designed to accommodate a variable number of functional elements of the system, usually including processing elements, I/O channels, and memory elements. This makes some performance measures dependent upon the configuration.

2.4 MEASURES OF PERFORMANCE

Measuring multiple processor real-time digital signal processing systems presents two problems not encountered in the measurement of uniprocessor general purpose systems. The first, which applies to most multiprocessor systems, is that each system has a unique program execution model. There are many variations in the size and structure of both instruction and data memory space and the interconnections between processors. This affects the task granularity of the application implementation and execution sequence of the tasks.
The second, which applies more to systems specialized in digital signal processing, is that there is not a single method for software development. Systems are usually programmed on multiple levels, with processing elements programmed with the lowest level of code, task coordination programmed at a higher level, and application control programmed at the highest level. The consequence of these variations is that a measure of the execution of an application or any set of tasks on one system may not be a valid measure on another system.

Providing some insight into this problem is a long term goal of the system measurement work described in this paper. The initial approach to system measurement which is described here is to divide the measurements on each system into three parts; theoretical peak, abstract operation, and application execution, as defined previously in section 2. Comparison of these measures provides valuable information about how an application is performing compared to its ideal performance (abstract operation) and to the full potential of the system hardware (peak operation). These results can be used to judge the suitability of a system to perform particular functions and particular applications as well as how well the system is being programmed and its potential to execute applications with similar structures. The application of this approach and the method of presentation of results is outlined in the following section.

2.4.1. Peak Measures

In this report peak measures are presented for each element of the system. These include a processing element, external I/O device, a global memory resource, and an internal data transfer resource. The architecture of each element is described on a high level. This is included with the performance numbers because peak measures are in essence a description of the physical structure of the elements. With the architectural features as a reference, the elements are described with performance numbers. Table 1 shows the format with which the measures are to be presented and lists the architecture elements and the measures associated with them.

2.4.2. Abstract Measures

Abstract measures are taken while the system is in operation under an artificially ideal mapping of functions to system elements. This is accomplished by programming each of the system elements to perform a single task at maximum possible utilization, without the need for synchronization or coordination of tasks.

By using the simplest control structure available to perform an operation such as a data transfer or an arithmetic computation, the abstract measurement provides a best case usage for the element against which to judge both the system control mechanisms and the application performance potential. If the abstract measure, for instance, is compared to the peak measure of the component, it indicates the cost of the system control mechanisms and gives a maximum measure beyond which the user cannot expect the application to perform. If the abstract measure is compared to the same measure in an application, it indicates the degree of success the application attains in using the system resources.

Table 1. Peak Measures Presentation Format

System Element	Processing Element	External I/O	Global Memory	Data Interconnect
Total peak operation				
Single operation rate				
Data storage capacity				
Instruction storage capacity				
Simultaneous functions				
Word width				
Maximum number of system elements				

The presentation of abstract measurements comes in two parts. The first is the operating overhead measurements, including data transfer times and task scheduling overhead. These numbers are presented with operating time lines, to help define the time intervals being measured.

The second is the throughput performance measures. The measures must be prefaced by a description of both the configuration of the system under test and the operations being applied to the system for measurement. The operations must be described in sufficient detail to understand the computations that are performed and on which element they are being performed. The timing intervals and the number of arithmetic operations per processing cycle must also be included. These numbers are presented in reference to the introductory description of the operations applied to the system. It includes for each operation under test the computation type, the number of arithmetic operations performed per cycle, the total throughput, the time in use, the operating speed, and operating efficiency.

The measures for data transfers and scheduling overhead are as follows.

Data transfer from one element to another:

Path set-up time - The time to configure the physical data path

Data network transfer rate - The speed of the transfer while the transfer is in progress

Aggregate point-to-point transfer rate - Total functional element to functional element transfer mode

Number of simultaneous transfer paths - The number of transfers occurring between all system elements at a given time

Total transfer rate - The sum of all simultaneous aggregate point-to-point transfer rates

Scheduling overhead:

Task-to-task schedule time - Time from task 1 completion to task 2 start-up, where execution of task 2 depends on completion of task 1

Other task schedule time - Such as function schedule time and data network blocking

Processing element utilization - The percentage of time the processing elements are operating

2.4.3. Application Based Measures

The application based measures are used to provide a measure of the system performance under real operating conditions. The value of each measure is limited in that it applies only to the application under test. However, comparing results with abstract measures and for several alternative mappings of the application to the architecture, important information about the effectiveness of the software development tools, mapping of applications to the system, and operating system efficiency can be learned.

Results of application based measurement must be prefaced with a description of the software development tools for programming the system and the application that is being programmed. The application should be chosen to be typical of those that are of interest to the target user community. The mapping of application functions to the system hardware must also be described. By demonstrating alternate mappings, the utility of the software development tools and the flexibility of the system control mechanisms are evaluated. The results are presented in the same format as the abstract results.

SECTION 3

COMMON SIGNAL PROCESSOR SYSTEM DESCRIPTION

3.1 INTRODUCTION

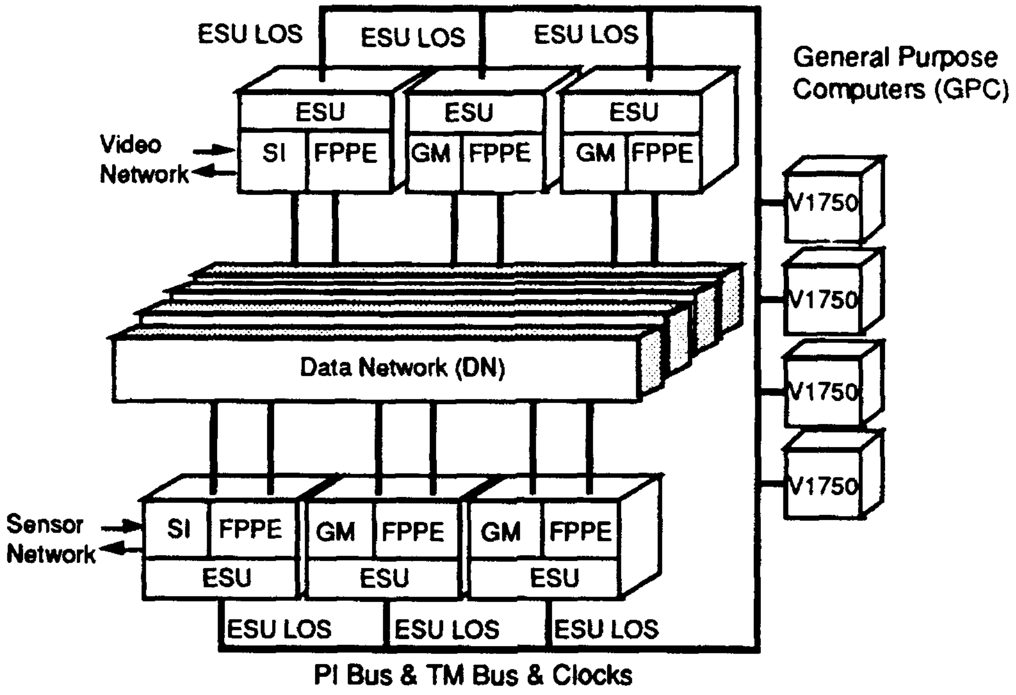
The Common Signal Processor (CSP) is a modular programmable digital signal processor developed by IBM Federal Sector Division to perform a wide range of high throughput real-time processing tasks, particularly radar signal processing. The processor fits the model of processors that are the target of this performance measurement investigation. To study performance of the system, the project obtained an agreement with IBM whereby the necessary equipment and support would be provided to MITRE by IBM. This section contains an overview description of the CSP architecture and its software development tools, a specification of the elements of the software development tools that were used to perform benchmarks, and a short scenario of the software development process for the CSP that was used to measure the system performance.

3.2 CSP ARCHITECTURE

The CSP architecture is partitioned into a core set of control and interface support modules, a set of functional element modules, and a set of support modules, as illustrated in figure 2 [7]. The support modules coordinate the functional element modules and support the transfer of data between them. Included with these is the Element Supervisor Unit (ESU) and the Data Network Element (DNE). The DNE is the building block for the Data Network (DN), which provides for data transfers between functional elements. The Local Operating System (LOS) residing on the ESU controls the functional elements and coordinates task assignment among them. Each ESU may control up to six functional elements.

Functional elements are specialized for the tasks of data storage, data processing, high speed I/O, and preprocessing. Modules that comprise current configurations of the CSP are the Floating Point Processing Element (FPPE), the Global Memory (GM), and the Sensor Interface (SI). The CSP is designed, however, as an open architecture and could accommodate modules to perform vector processing, specific preprocessing, or any specialized task required by new applications. For example, IBM is currently developing a Vector Processing Element with higher throughput vector processing. A brief description of the FPPE, GM, and DNE architectures is provided in section 4 along with their peak performance measures.

The set of support modules provides auxiliary functions to the system, including system I/O, user console interface, and general purpose computing. Application command programs are executed by General Purpose Computer (GPC) modules. They perform CSP processing mode control, generate sensor control parameters, and communicate with external systems. The system can be configured to match the requirements of the application, thus avoiding unneeded overhead and system cost.



FPPEs - 150 MFLOPS peak performance
Global Memory - 4 MBytes storage
Data Network - 25 MHz, 32 bit data paths

Figure 2. Common Signal Processor Architecture High Level Block Diagram (Courtesy of IBM)

The system execution model is shown in figure 3. The figure depicts a data flow graph, which is used to describe all signal processing computations in the CSP. In this graph, a data task is represented by a circle and involves data movement, i.e., computation or I/O. Between each task, data is stored in a storage object, which is represented by a box. The system execution model allows multiple tasks to be performed in parallel on data from one storage object and requires that all tasks operating on data from that object be scheduled simultaneously. All program specification for data computations is in the form of these data flow graphs.

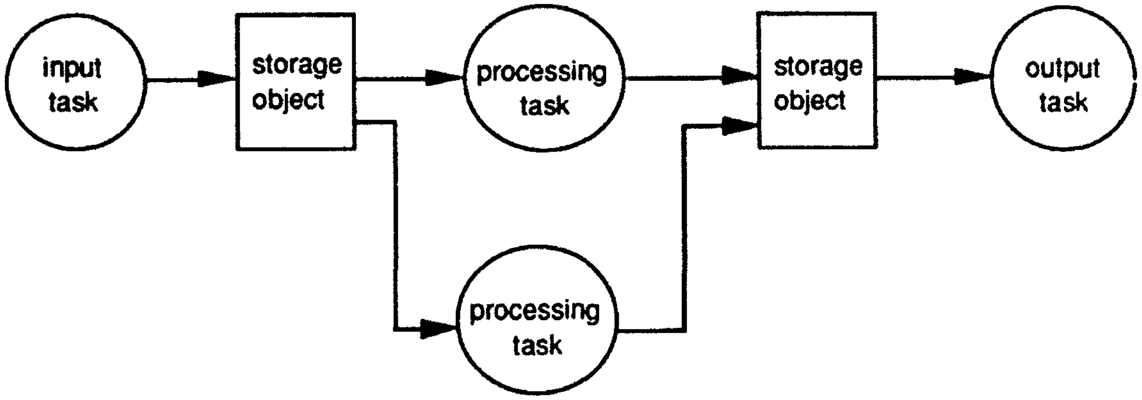


Figure 3. CSP Data Flow Graph System Operation Model

A data flow graph is executed by the system at a coarse grain level in an event driven fashion as depicted in figure 4 [5]. In the scenario shown in the figure, data arrives at a constant rate at the Sensor Interface element (SI). Data is transferred from the SI to Global Memory (GM) A when two conditions have been reached: the buffer threshold in the SI is reached, and space is available in GM A. After the last word of the block has been transferred, a *produce* signal is sent to GM A, signifying that the transfer is complete. The GM updates its data pointer and sends a *read enable* signal, signifying that data is available, to the down stream Floating Point Processing Element (FPPE). When data is available and the *write enable* signal has been received by the FPPE, signifying that space to write the computed results is available on GM C, the computation task is scheduled. In terms of the graph execution, the reading of a block of data from GM A, performing the data computation task on the FPPE, and writing results to GM C down stream is one event. At the completion of this event, the *produce* and *consume* signals are sent to the receiving and sending storage objects, GM C and GM A, respectively, causing them to update the data pointers. The storage elements may then generate new *write enable* signals to allow the cycle to repeat itself.

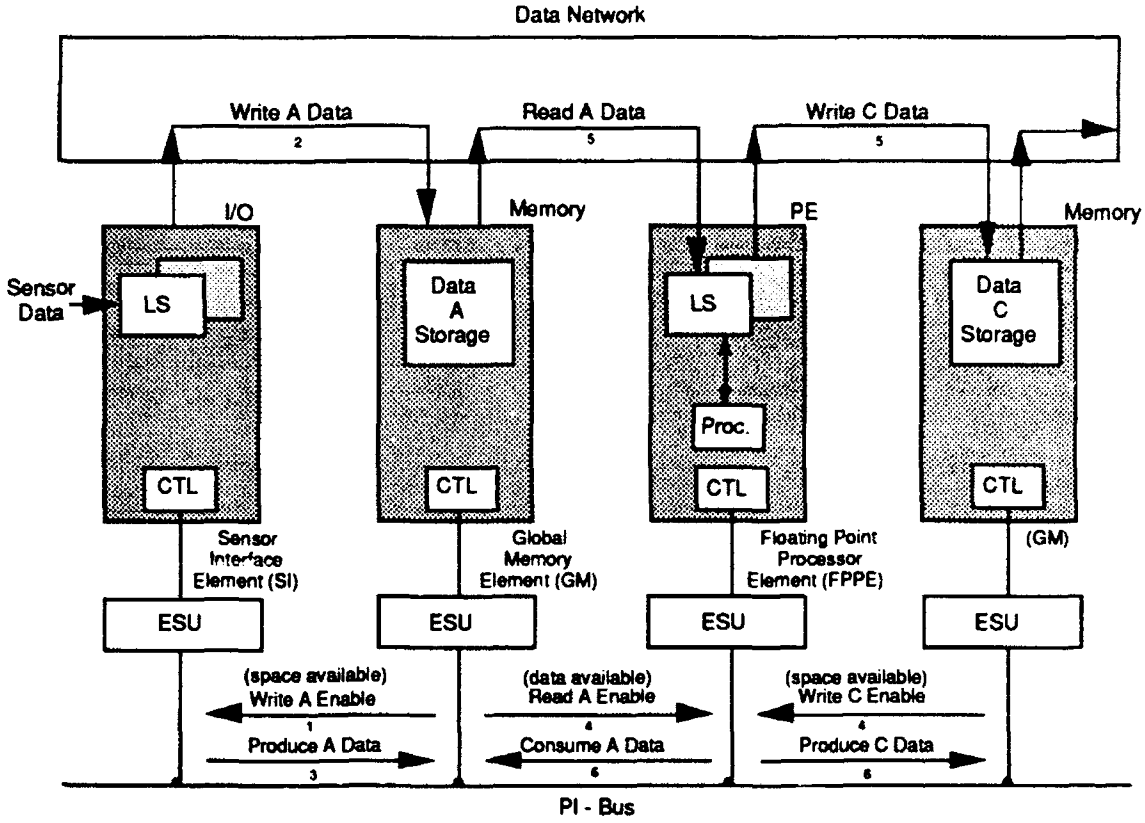


Figure 4. Data Flow Event Driven Execution Mechanism

3.3 THE CSP SOFTWARE DEVELOPMENT TOOLS

There are three levels of software development on the CSP, as depicted in figure 5. Each level of programming represents a different set of control functions in the system. Application Command Programs control the system I/O, data processing, and some high level aspects of the signal processing task allocation. Specific processing modes are programmed from a data flow graph description of the application, denoted as CSP graphs, that specifies the functions of the Element Supervisor Units. Computation tasks are defined by the CSP primitives and execute on the Functional Elements. They consist of macro program strings that are programmed at a microcode level.

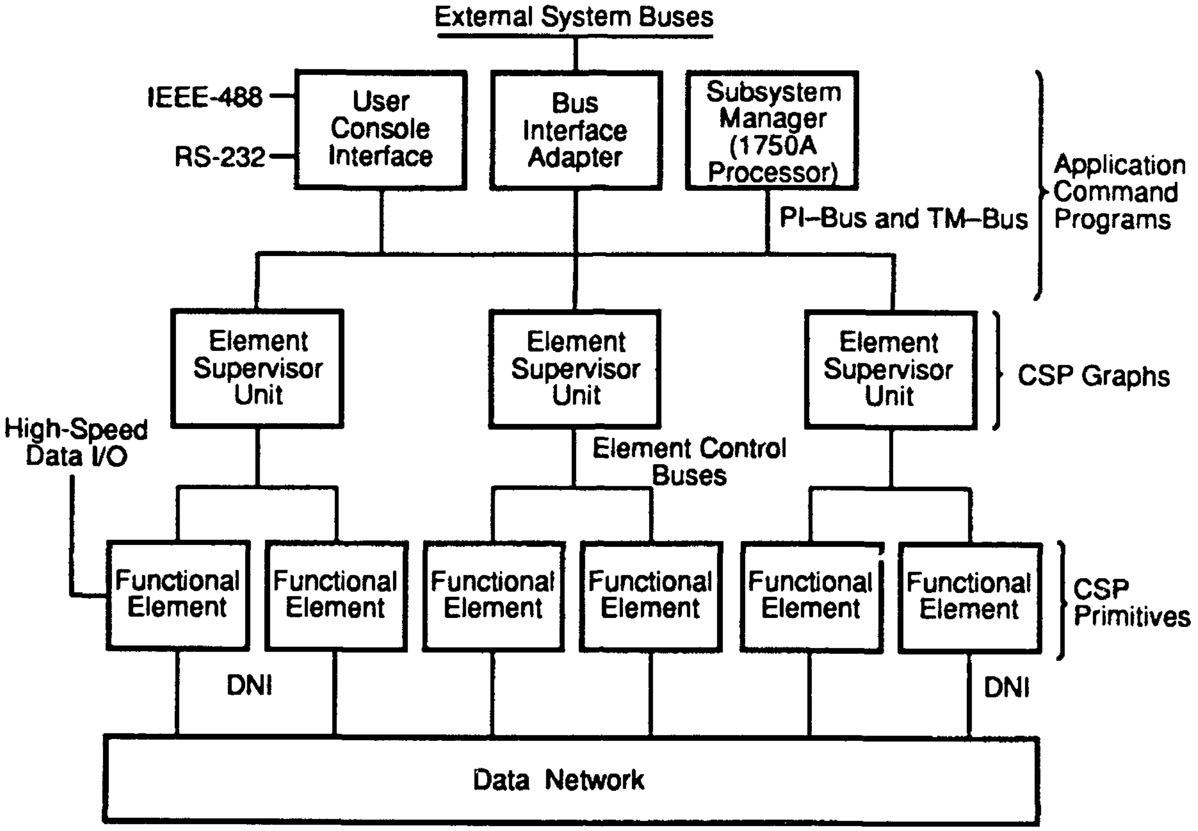


Figure 5. CSP System Software

The set of software development tools used to develop CSP software is shown in figure 6. On-line allocation of system resources and coordination of graph execution is controlled by the Subsystem Manager running under the Local Operating System, and programmed with a Command Program in the Ada programming language. The Element Supervisor Units control the operation of the FPPEs and other functional elements and are programmed with the Graph Program Preparation tools. The System Level Simulator is available to assist in developing effective data flow graphs. The FPPE computations are specified with individual FPPE macros, which are written in microcode with the assistance of the FPPE micro assembler and simulator and are stored in a macro library. The macros include data buffer specifications and execution time information required by the simulator and complete FPPE micro instructions. Micro instructions are incorporated into the executable graph binary image by the Graph Linker. The three levels of programming are combined by the System Build component, which creates an executable binary load image of the application that combines and coordinates the three levels of operation of the full system. IBM loaned to MITRE the tools required to execute a data flow graph on the System Level Simulator. These are indicated in figure 6 with a stipple pattern, and include the Graph Translator, Simulator, and the library of FPPE macro files.

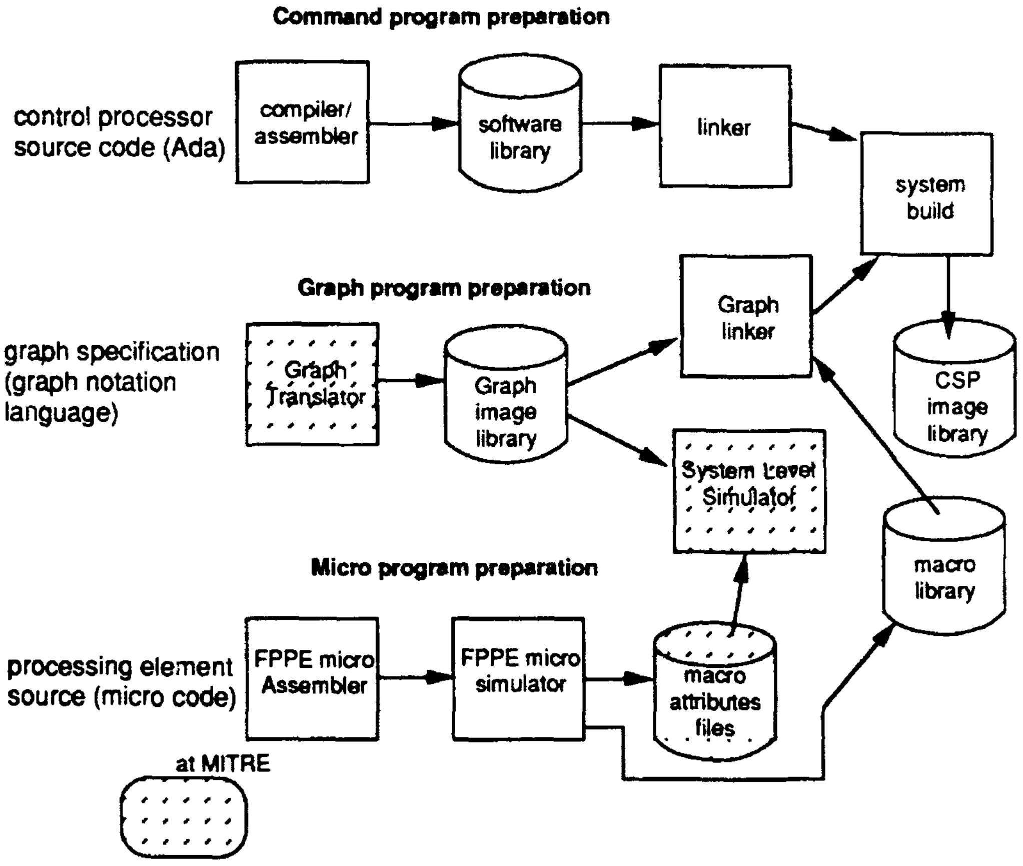


Figure 6. CSP Software Development Tools

3.4 DEVELOPING CSP SOFTWARE

Programming a CSP application begins with the specification of a data flow diagram, such as the generic example shown in figure 3. The user further specifies the signal processing tasks of the application and defines them in a graph description language to the Graph Translator.

Each task is the complete processing on one FPPE that is executed on one set of data. In order to describe the task in the graph description language, the programmer must specify the task at the level of detail shown in the example in figure 7. The figure shows the access to Global Memory, the exact placement of the data in the local memory, and the computation macros to be executed. Issues such as overlapping input and output buffers and computation times are dealt with at this level. From this specification, the programmer can make a good estimation of task completion time and map the tasks to the system resources according to the computation and data storage capacity of the FPPEs and the GMs.

The Graph Translator translates the graph and its mapping to a series of events to be executed by the system. This event sequence is the input to the System Level Simulator that simulates system performance and generates detailed timing information to reflect the expected system execution of the graph.

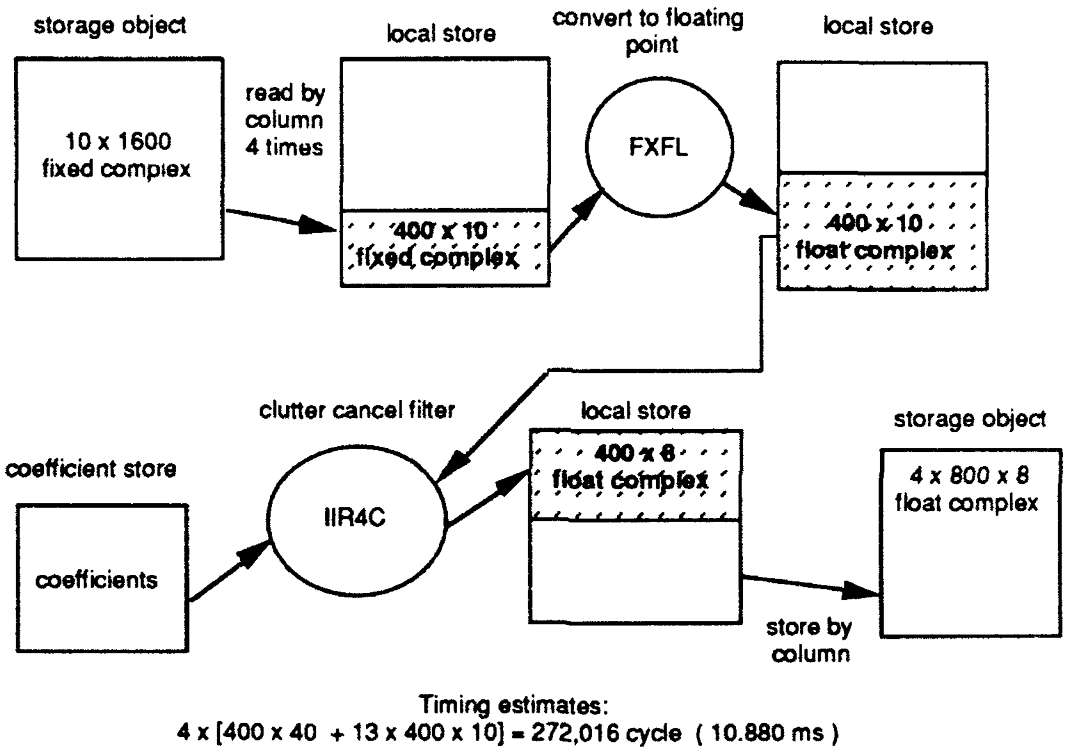


Figure 7. Task Specification for the Graph Translator

Writing computation macros is a separate development process. An extensive library of previously written and tested FPPE macros exists, including functions such as an N-point FFT or a correlation. If additional macros are needed, they are written in microcode and translated to machine code by the FPPE micro assembler. Routines are tested on a detailed hardware model of the Floating Point Processing Element (FPPE) that simulates data results and precise timing characteristics. The results of this simulation are used by the System Level Simulator to provide an accurate simulation of the graph execution of the application.

The third section of software development is writing the Application Command Program. This aspect of the software controls the system interface modules and data dependent control functions. Once the FPPE macros and graph design have been fully tested on the simulators, the Application Command Program is used to develop the complete executable program for real-time on-line execution.

3.5 DEVELOPING BENCHMARKS USING THE CSP TOOLS

The benchmarking software development has been conducted at MITRE, Bedford, without the system hardware. As stated above, IBM loaned to MITRE the elements of the software development system required to make extensive software development and system performance characterization possible. The System Level Simulator executes a behavioral simulation of the system at the event level with the execution specified with the Graph Translator graph image.

The Graph Translator is written in a behavioral simulation language called Simscript. It runs on a VAX VMS system and requires the Simscript runtime library. IBM obtained permission from CACI, the company that licenses Simscript, to include the library with its tool set at no extra cost. They also included a graphical time line display tool (TML) for examining the simulator output. TML runs on a PC, and includes a data translation filter to give it compatibility with the VAX VMS.

The Graph Translator comes with an extensive library of FPPE macros that has been developed by IBM. The macros perform such operations as format conversion, filtering, threshold comparison, FFTs, and other operations needed to construct radar processing applications. Although the simulator does not actually execute the macros, it does depend upon fully developed code for complete data I/O specification and the parameterized performance equations. According to the IBM applications engineering staff, the average macro takes four staff weeks for experienced staff to develop and would cost about $15,000 each if IBM were contracted to develop new macros. Due to the completeness of the existing set of macros provided in the macro library, the benchmarking proceeded to test CSP performance without the extensive work or expense of developing new macros.

A data flow graph is encoded in a graph description language. The Graph Translator transforms the description file into a set of control specifications that fonn the basis for the input to the System Level Simulator. The System Level Simulator simulates the execution of the data flow graph and produces detailed information about the system performance. Attributes and performance of the FPPE macros are described in detail to the simulator in a set of attribute files. The attribute files contain performance information that has been verified with physical system measurements and allow the simulator to reflect accurately the performance of the FPPE computations.

The output of the simulator includes a detailed set of statistics describing the system operation. Each event in the execution of the graph is listed in both time order and sorted by functional element. Statistics are also provided on the execution times of the FPPEs and the status of data storage in the Global Memories. Times are given to microsecond accuracy. The time line display capability is very helpful to gain an understanding of the system operation. TML displays the graph on the screen, including any or all of the functional elements and zooms or pans to provide any level of resolution. It also shows task execution and control signals. These graphical displays were used to find critical sequences and durations, and the simulator file output was referenced for exact timing measures.

SECTION 4

RESULTS OF COMMON SIGNAL PROCESSOR PERFORMANCE MEASUREMENT

4.1 INTRODUCTION

As outlined in section 2, measures of performance can be seen from three perspectives and each makes its own contribution to understanding the system. This section is presented in the format established in section 2, and describes the performance of the CSP. Some practical considerations that were required to proceed with the work are given here as an introduction, and the following three subsections present each of the three levels of measurement that are described in section 2.

The first step of benchmarking the CSP was to install the Graph Translator and TML tools, establish a working system and learn to operate it. Some support from IBM was provided to help us learn to use the Graph Translator and associated tools and included a three day course given by a CSP system expert. The course included extensive class notes detailing system operation and use. The peak measures were compiled from the information presented in the course, and provided a preliminary understanding of the system operation. A realistic application was then programmed, and served as the vehicle for learning to use the software tools. The application was programmed on the CSP by transcribing its data flow description to the CSP graph notation language. The system simulation of this graph was used for making application level measures of the system. A separate data flow graph was then developed for abstract measures. These measures were optimized with the benefit of the understanding of the system operation gained with the previous set of measures.

All processing element computation performance numbers presented here are given in MFLOPS, and refer to the number of million adds, multiplies, or conversions from integer to floating point numbers performed per second.

4.2 PEAK MEASURES

4.2.1 Processing Element

The FPPE is a micro-programmable signal processor with local storage for program code and data. A functional block diagram of the FPPE is shown in figure 8 [5]. There are two data buffers called Local Store A and Local Store B, both with connections to the Data Network (DN), allowing the execution and input/output operations to overlap. A series of FPPE macros may be stored in the Microstore and chained together to form a macro string which operates on the data in local store. The FPPE contains two parallel pipelined data paths, each with a multiplier, accumulator, and an ALU. Coefficients may be read from a coefficient store or generated by a function generator capable of performing sine, cosine, arctangent, square root, and reciprocal functions in two clock cycles. Table 2 provides the peak performance parameters of each of the CSP functional elements and the data network.

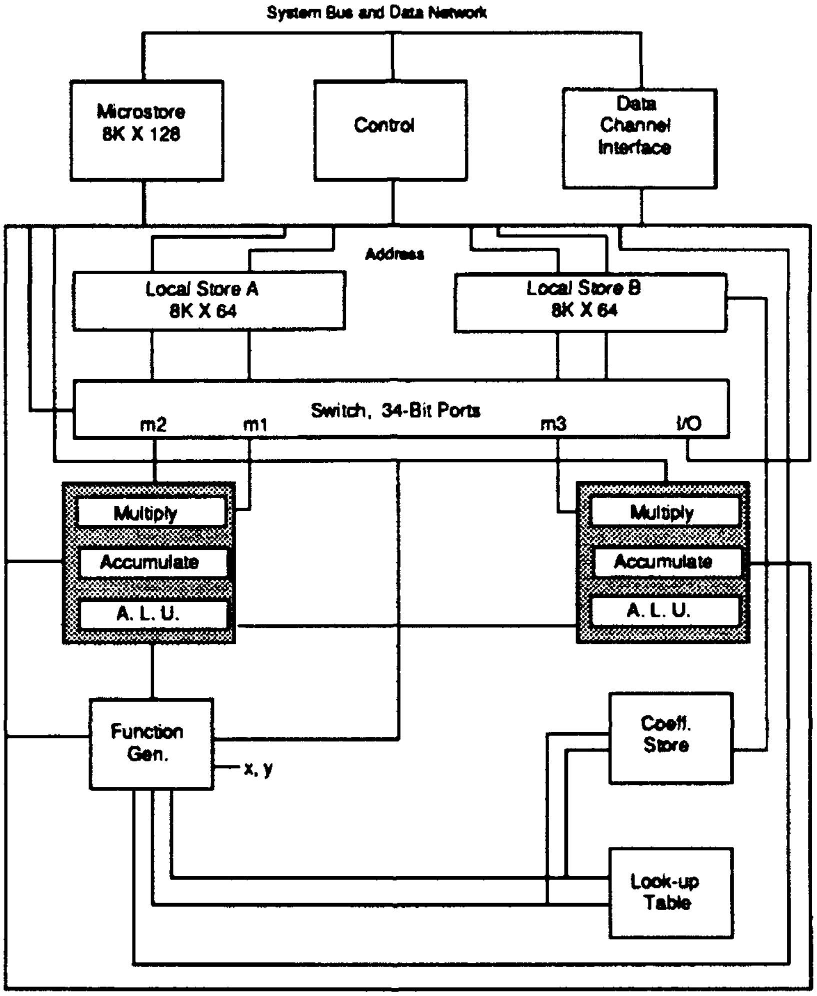


Figure 8. Floating Point Processing Element

4.2.2 Sensor Interface

The Sensor Interface (SI) is a functional element designed explicitly for the transfer of data between the CSP and external data sources. The SI, shown in figure 9 [5] , operates in either point-to-point or local area network mode. The transmit and receive channels of the interface are each capable of supporting data rates up to 400 megabits/second. The Local Stores (LS) allow for reformatting of data before transmission. One LS is dedicated to each input and output, and the third may augment the buffering for either the input or the output, as required. Transfers across the Data Network are performed in blocks, as specified by control signals from the ESU. Peak performance parameters are shown in table 2.

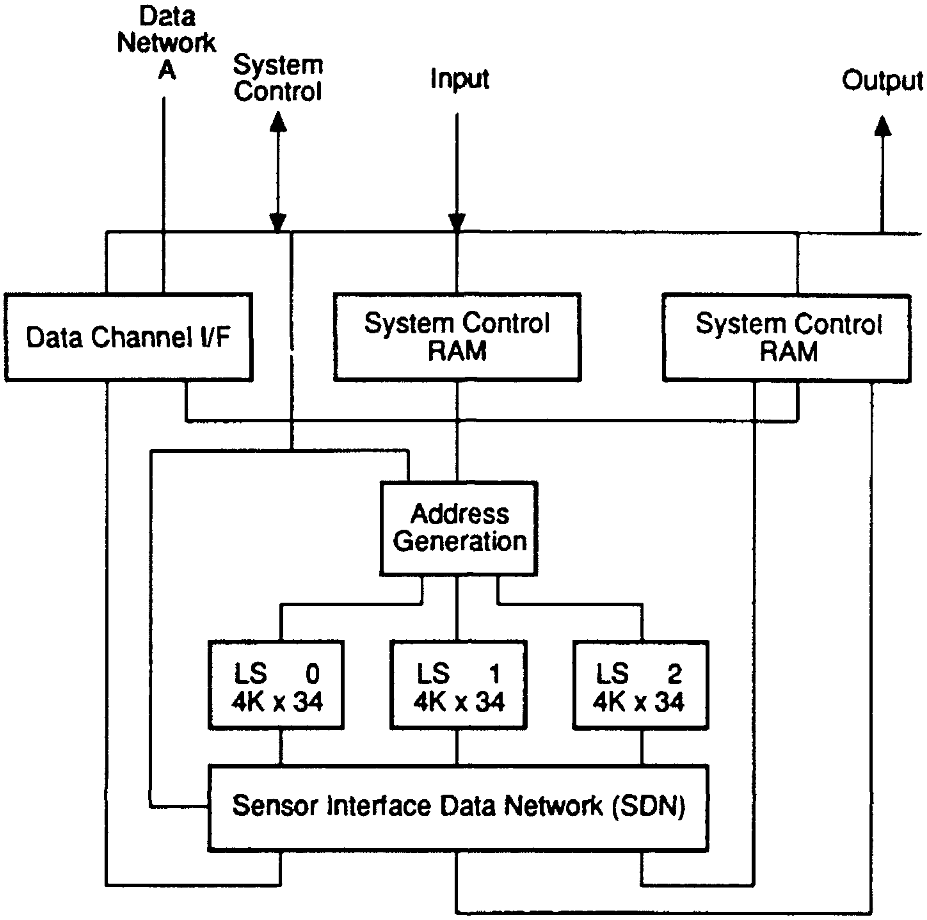


Figure 9. Sensor Interface High-Level Block Diagram

4.2.3 Global Memory

Data storage within the system is provided by the Global Memory (GM) modules. The structure of the GM is shown in figure 10 [5]. Processing elements or I/O modules under the control of any ESU in the system may access the storage objects assigned to a GM via the Data Network. Data storage is formatted in 32-bit words and an additional 8 bits for error correction and detection. The module supports several addressing modes, which allow the operating system to treat storage objects as either queues or buffers. A buffer type storage object is an unstructured organization that is accessed by explicit reference to buffer offset addresses. A queue type storage object is organized as a circular queue, where data is written sequentially to the tail of the queue and read sequentially from the head of the queue. The data is described in terms of a matrix, and, using the queue model, can be accessed as a linear queue, as multiple queues, or as a matrix in corner turn or coordinate rotation modes. Global memory peak performance parameters are given in table 2.

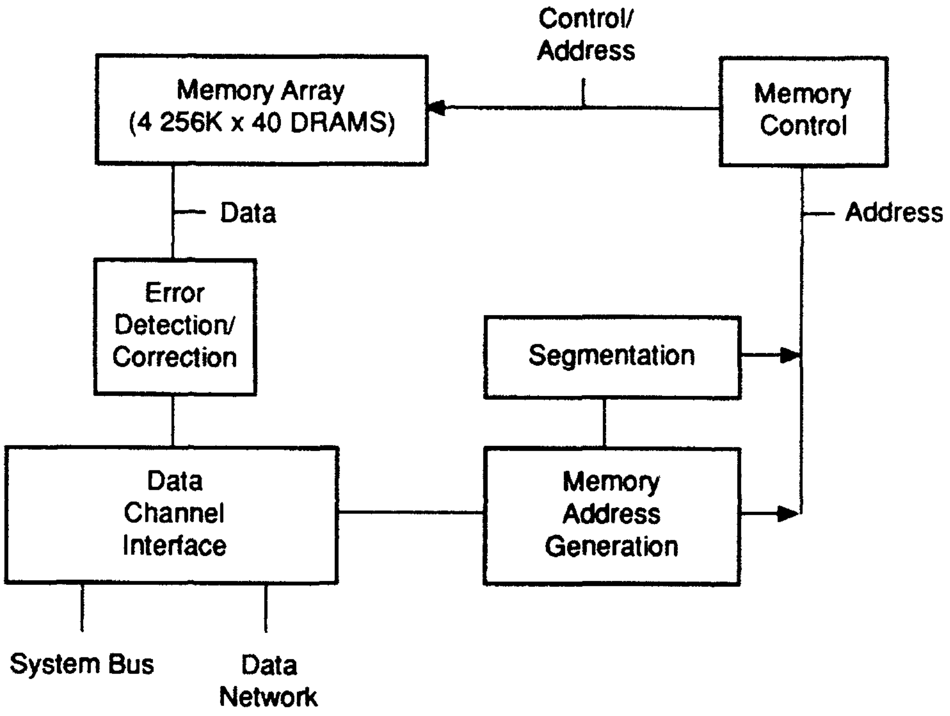


Figure 10. Global Memory Functional Block Diagram

4.2.4 Data Network

The Data Network provides a high-speed data transfer mechanism between functional elements, and is configured in either 16- or 32-bit wide paths. The size of the Data Network depends on the number of Data Network Elements (DNE) used. A single Data Network Element, made of eight switches, is shown in figure 11 [5]. Each dataline in the figure represents 8 bits and the combination of Data Network Switches comprises a single DNE with six 32-bit wide ports. The maximum network size is limited only by the physical packaging constraints of the system backplane and a limit of 8 switching nodes in any one data path between two functional elements. The number of simultaneous transfers possible depends on the interconnection scheme chosen and the maximum number has yet to be fixed. Routing of a path between two functional elements occurs for each data block transfer and is based on a set of route addresses associated with a block header that contains one route address per switching node in the path. Links in the network may be shared by multiple paths, and are assigned at graph compilation time. Peak performance parameters are shown in table 2.

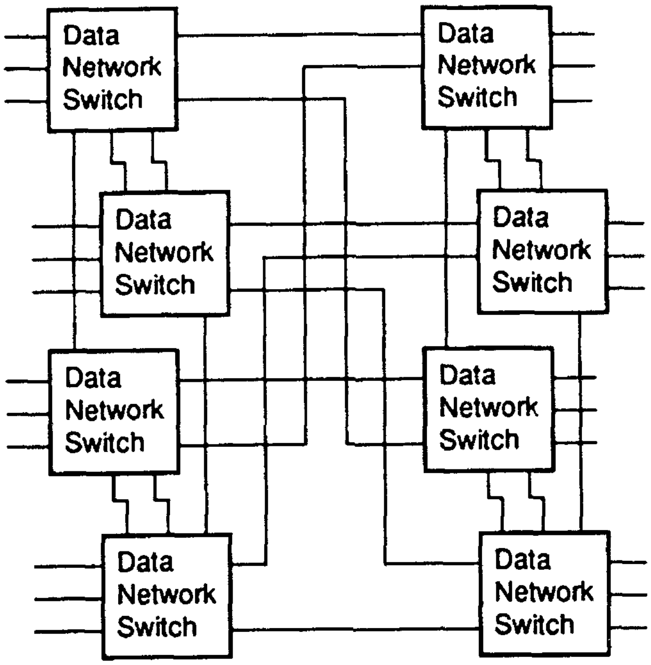


Figure 11. Data Network Element

4.2.5 System Configuration

The numbers in table 2 are representative of a realistic configuration on the largest existing backplane (78 slots total). The actual configuration may be customized and future packaging technology will allow larger systems to be integrated.

4.3 ABSTRACT MEASURES

Results presented in this section are based on system-level simulations of a CSP configuration with 5 FPPEs, 4 GMs and 2 SIs. The simulator timing estimates have been shown to be accurate by comparison with hardware performance times. The simulations are derived from software code that is identical to code that would be prepared for running on the system hardware.

Abstract measures are performed on the CSP with the data flow graph depicted in figure 12. Each processing element in the 5 FPPE configuration was programmed to perform a different computation task. The tasks are typical of those used in radar processing. The graph represents one cycle of execution, and a node on a data flow graph represents the processing to be performed on one FPPE during a cycle. At the end of each cycle the output of each node serves as the input to the next node in the graph to be processed at the next cycle. Each task is sized to match the graph cycle time in order to minimize scheduling wait time. The application is artificial in the sense that the task sizes are determined by the time available in the cycle rather than the requirements of the data. A real application would have no such flexibility.

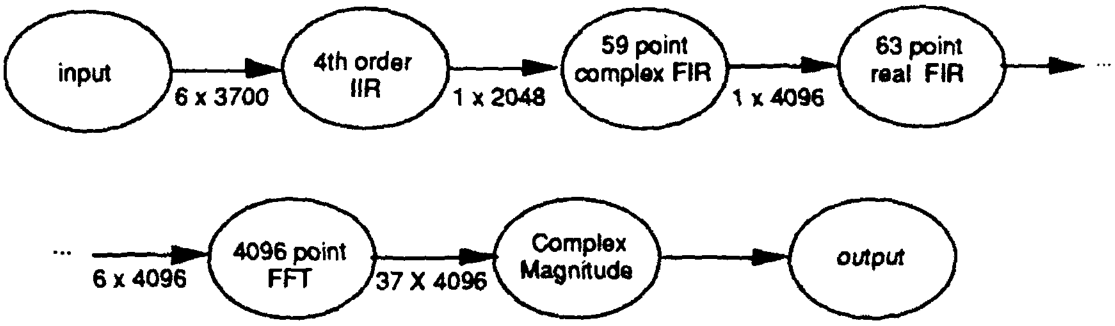
Table 2. Peak Measures of the CSP

System Element	Processing Element	External I/O	Global Memory	Data Interconnect
Total peak operation	150 MFLOPS	400 Mbits/sec	800 Mbits/sec	1200 Mbytes/sec
Single operation rate	25 MHz	12.5 MHz	25 MHz	25 MHz
Data storage capacity	64 kbytes	48 kbytes	4 Mbytes	none
Instruction storage capacity	128 kbytes	--	--	--
Simultaneous functions	6	2 (in&out)	1 (read or write)	12 (estimated)
Word width	8-64 bits	16-32 bits	16-64 bits	16-32 bits
*Maximum number of system elements	12	4	8	8

*reflects largest configuration existing to date

The cycle time for the data flow graph in figure 12 was determined by the FFT task. It is the largest possible FFT that can be executed with the available library macros, due to the local memory size constraint of the FPPE. It is repeated six times in the task, to allow an easily measurable amount of processing to be conducted during each cycle. The other tasks were sized by adjusting the data set input size and the number of iterations to be performed in one cycle, with an effort to use the largest possible input data set allowed by the internal FPPE memory size. Each of the tasks are shown on the data flow graph with the function name and its input parameters in the circles. Between tasks, the numbers under the arrows show the number of times the tasks are repeated in one cycle by the size of the input data set for each repetition.

The graph was entered to the Graph Translator in the graph specification language according to the mapping to the system shown in figure 13. There are no complications to the mapping because the graph was developed with this mapping as the model of execution.



Typical radar computation tasks - sized to balance computation time at each node

Figure 12. Abstract Measures Data Flow Graph

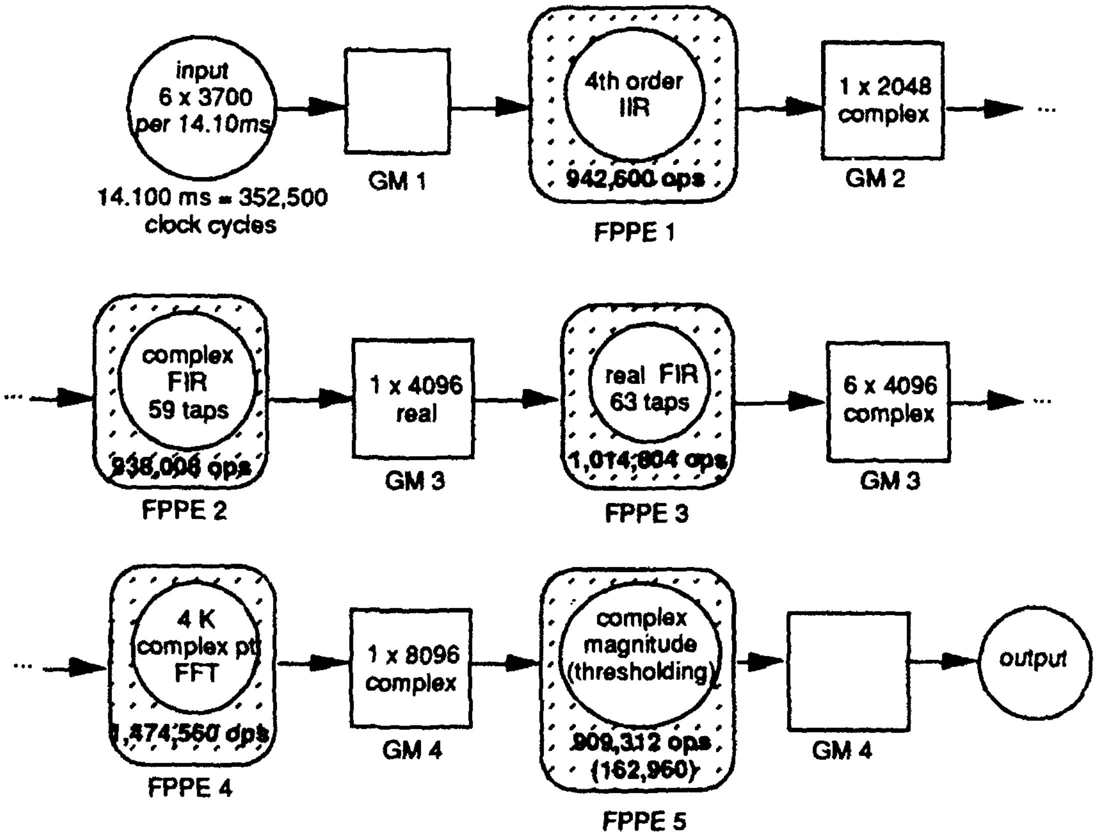


Figure 13. Mapping of Abstract Measures Data Flow Graph to CSP

Simulator execution of the abstract measures flow graph results in a detailed set of time stamps that represent system operation. This operation is displayed on time line graphs that can be viewed at any level of detail to illustrate all aspects of the system execution. Figures 14 through 18 show key features of the abstract measures data flow graph execution. Figure 14 is the highest level view, and shows the initial set-up and four full cycles of graph execution. Each horizontal line of the graph corresponds to one element of the system shown in the application mapping in figure 13. Execution begins with data coming in at Sensor Interface (SI) 1 which is illustrated in the timeline by a rectangle whose width represents the time it takes for the input data to arrive. Between each task execution cycle, the data is transferred to Global Memory. The low bars on the GM lines represent loading of data into the memories, and the high bars represent unloading the data, as it is transferred to the next FPPE task. Output begins at SI 2 after five cycles, when the PE 5 executes for the first time.

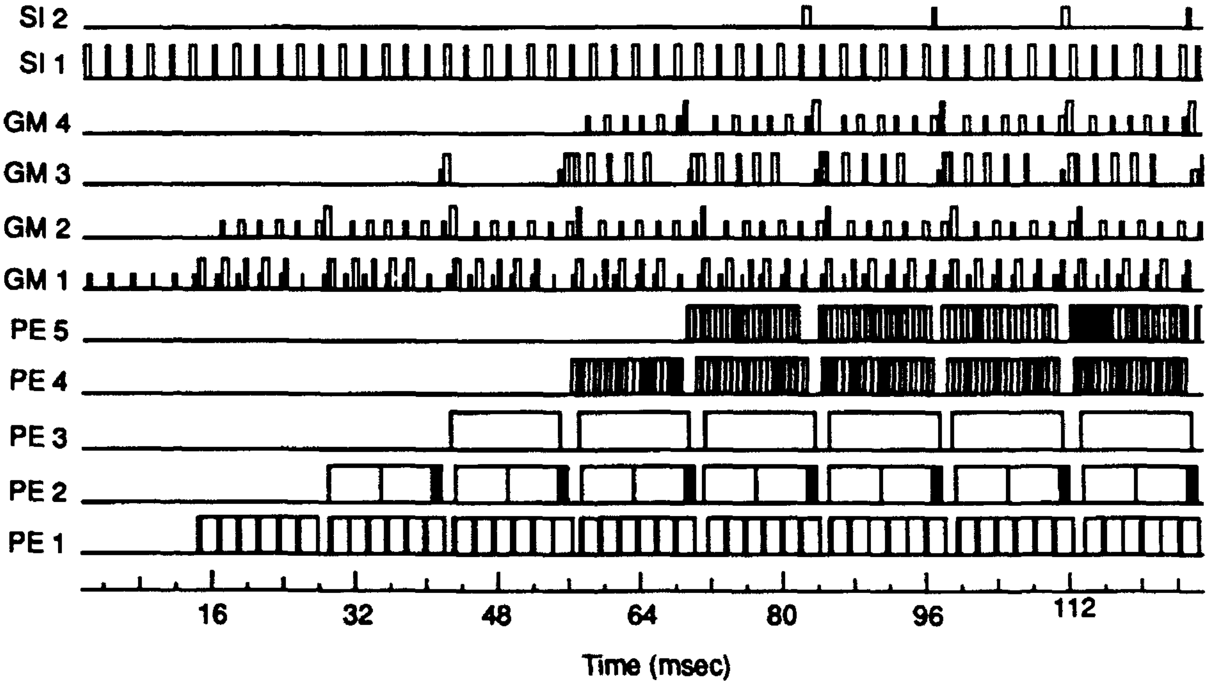


Figure 14. Maximum Throughput Execution Timeline

4.3.1 Data Transfer

Viewing this graph in more detail reveals more about the task scheduling overhead and data transfer times. Figure 15 focuses on the events between the completion of a task on FPPE 4 (PE 4 on the graph) and the beginning of a subsequent task on FPPE 5 (PE 5). At the completion of the last subtask on FPPE 4, the DN connection is established and the data is transferred to the GM. The GM overhead time includes reading control signals, updating data pointers, and sending ready signals to the ESU. When the ESU schedules the next task, the data transfer begins to load FPPE 5. Critical times shown in table 3 are labeled on the graph.

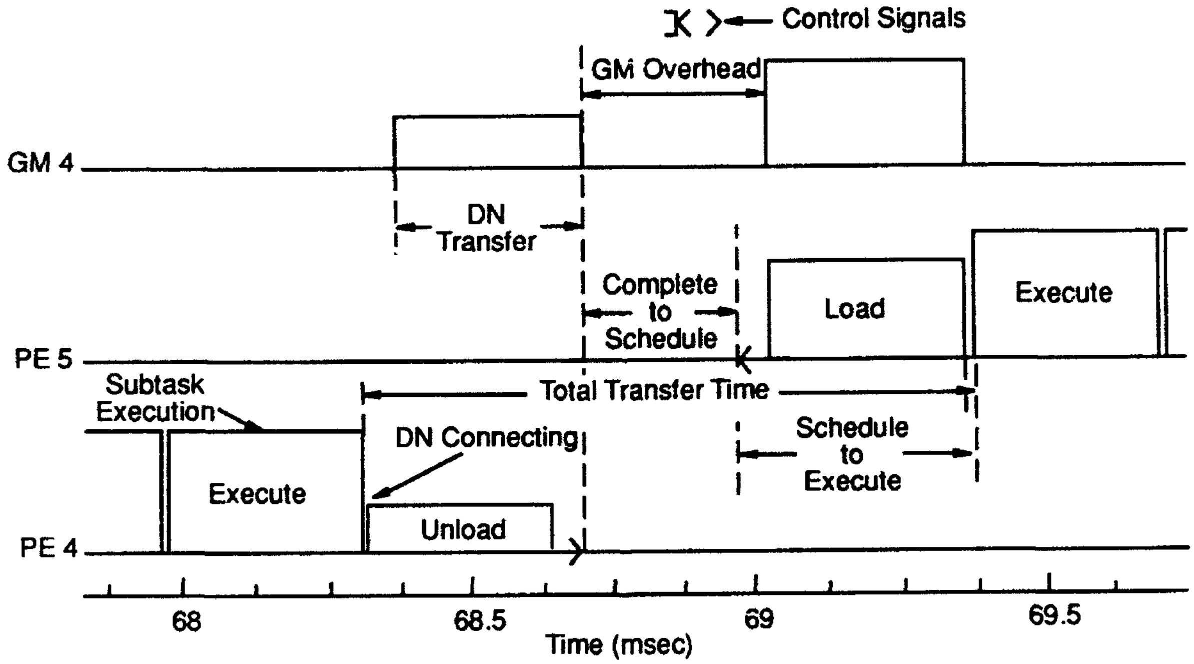


Figure 15. Task Scheduling and Intertask Data Transfers

Table 3. Data Network Measures

Path set-up (DN connecting)	2-3 μsec
Data network transfer rate	24.5 MHz (784 Mbits/sec)
Storage object unload to load interval (GM overhead)	479 μsec
Aggregate point-to-point transfer rate (functional element to functional element)	7.1 MHz
Number of simultaneous transfer paths	5
Total transfer rate (maximum number of simultaneous paths was not used)	35.5 MHz

The graph execution in figure 14 displays no access conflicts at the storage object. It is important to note that storage objects may be a shared resource for some applications and configurations. In such a case, processing element task scheduling delays can result from contention for access to storage object data because there is only one storage object port to the data network. To illustrate this problem, the task mapping shown in figure 13 was changed, so that tasks on PE1, PE2, and PE5 all access GM4 for data input. The resulting execution of this modified mapping is shown in the timeline in figure 16. The time delay in the sixth task execution of PE1 shows that contention for data from GM4 has caused a scheduling delay.

The graph also illustrates how the delay is compounded by the scheduling requirement for an *output data space available* signal. The task on PE3 does not access GM4, but is delayed on its sixth execution because the *consume* signal from the task on PE4 is required before the *space available* signal from GM3 is generated. This example illustrates how inflexibility of the scheduling requirements must be accommodated by ample memory resources in the system configuration and data flow graph mapping, and how the data transfer mechanisms depend on the Global Memory resource.

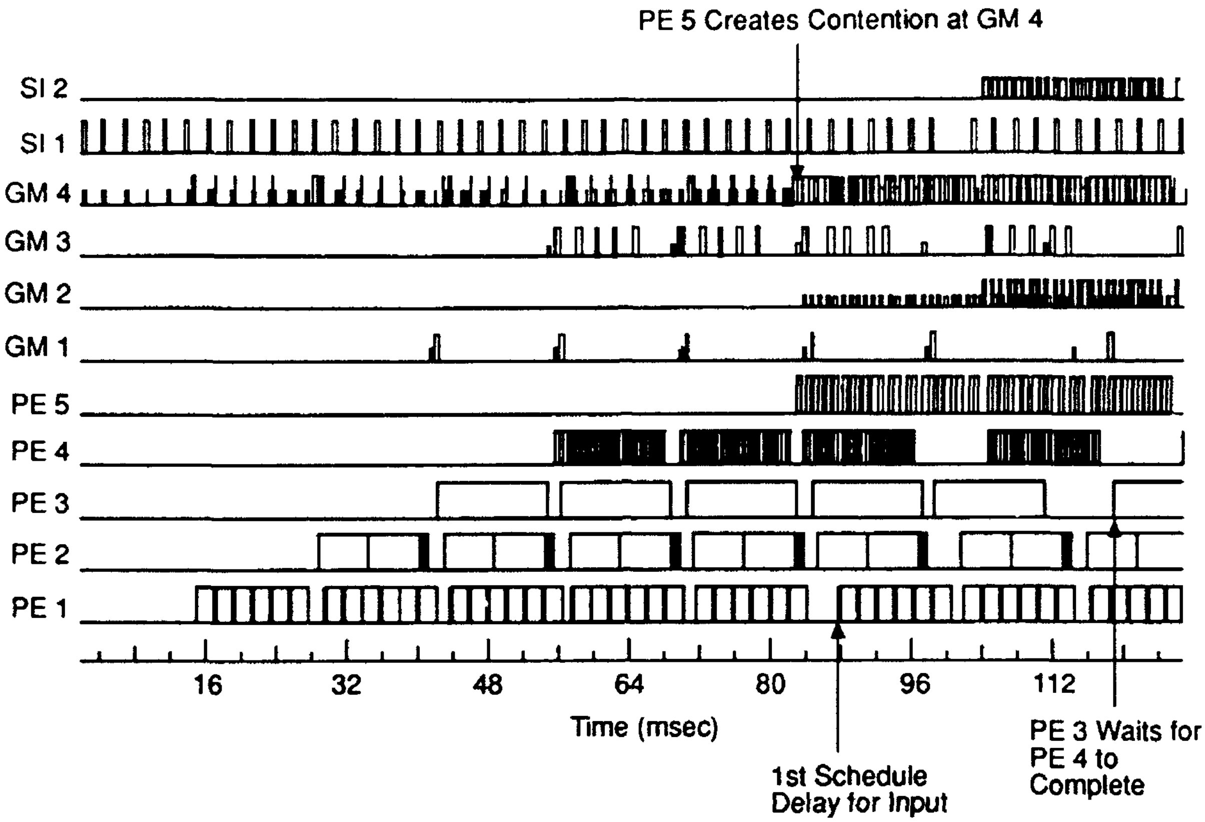


Figure 16. Memory Contention Problems Example

4.3.2 Scheduling Overhead

Figures 17 and 18 focus on the details of task execution from figure 14. A computation task begins with a *schedule* signal from the ESU. This is followed by loading of data from the storage object to the processing element. Each task consists ot suotasks that are executed repeatedly for a prescribed number of times on a new set of data from the storage element, and each subtask is followed by unloading of result data to a storage object. Each subtask consists of a string of macros, and is constructed to operate on data from the processing element's internal data buffer.

Figure 17 displays one full cycle of task execution (on PE 4), including the task scheduling signals, the input and output of data to the GMs (GM 3 and GM 4), and initiation of the next task in the processing chain (PE 5). The times given in table 4 are labeled on the two graphs. An interesting feature of the task execution is shown more explicitly in figure 18, which is a more detailed look at the subtask execution. The subtask scheduling overhead does not depend on the input and output of data, because I/O takes place during subtask execution. This is made possible by the use of the dual local store buffers on the FPPE.

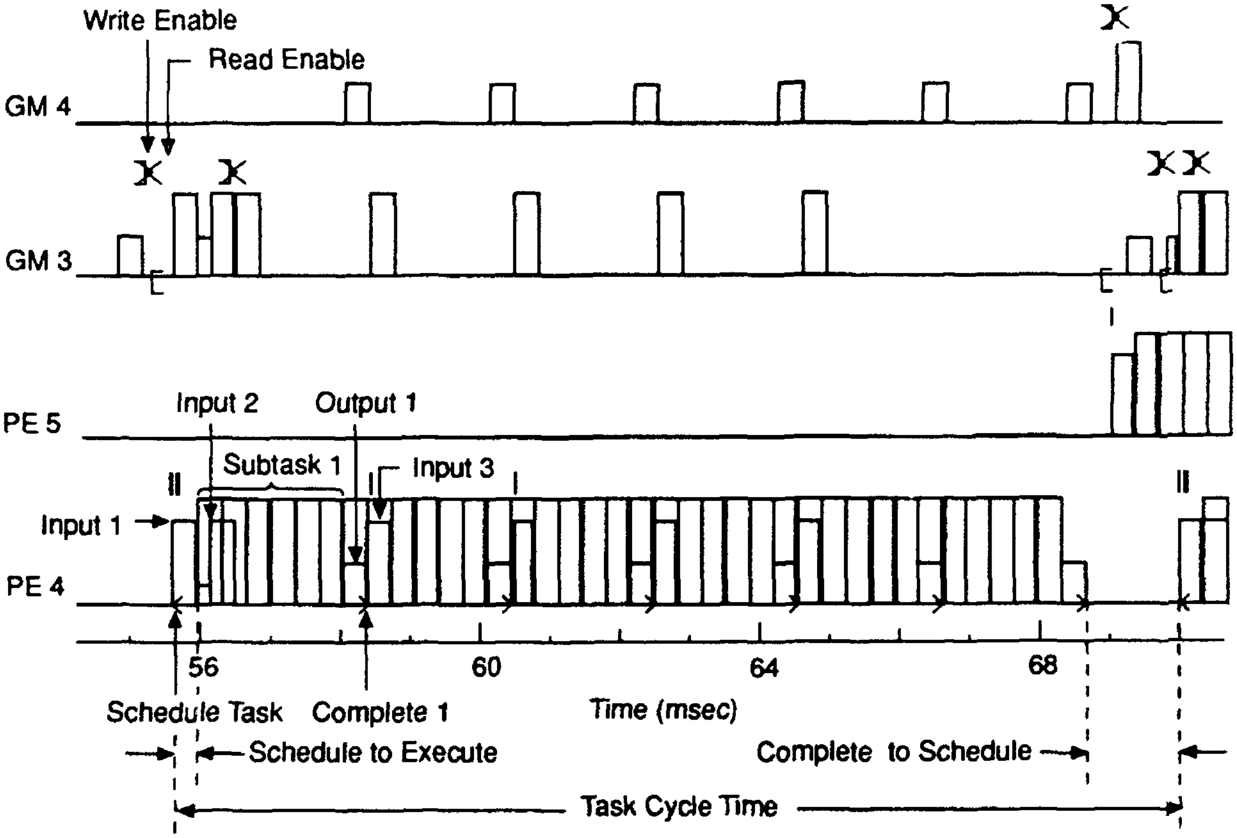
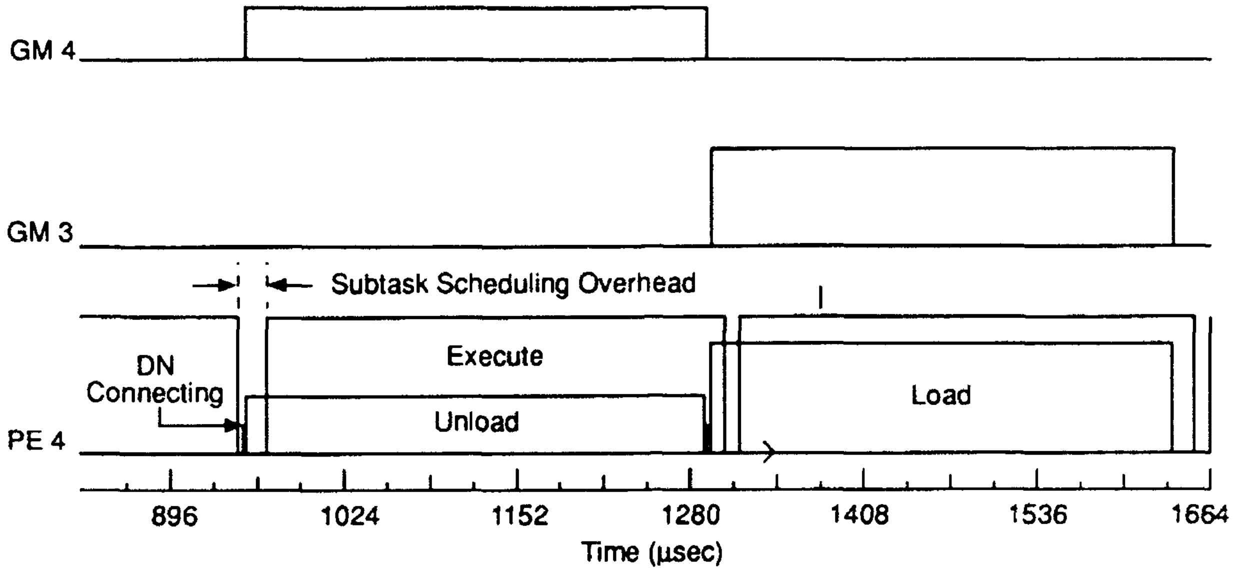


Figure 17. One Complete Task Execution



Subtask to Subtask Execution Time – 20 µsec

Figure 18. Subtask Scheduling

Table 4. Scheduling Overhead Times

Schedule to execute	393 - 397 µsec
Complete (task 1) to schedule (task 2)	287 - 458 µsec
Subtask complete to next subtask execute (subtask scheduling overhead)	20 µsec
Processing element utilization	86 - 88%

4.3.3 Processing Element Computation Rate

The computation rates summary presented in table 5 draws on information from each of the graph details shown above. As is shown, the rates obtained varied according to the task being performed. The second column in the chart is the total number of floating point operations performed during the computation cycle. The third column is the total computation throughput attained in the cycle, derived by dividing the number of operations by the cycle time. The next column gives the total time in use as a percentage. This is given to describe the system's scheduling efficiency and is derived by the total time spent in task execution divided by the cycle time. The next column is the operating speed, and shows throughput of the processing element while it is actually executing a computational task and is given by the throughput divided by the percent time in use. The last column, operating efficiency, compares the throughput to the best possible device usage by dividing the operating speed by the peak throughput.

Table 5. Processing Element Abstract Performance

Task	Number of Operations	Total Throughput MFLOPS	Time in Use	Operating Speed MFLOPS	Operating Efficiency
IIR	942,600	66.8	89%	75	50%
Complex FIR	938,008	66.6	90%	74	49%
Real FIR	1,014,804	72	88%	82	55%
4K FFT	1,474,560	104.6	87%	120	80%
Complex Magnitude	909,312	64.5	90%	71.6	48%
Thresholding	162,960	11.3	95%	11.9	8%

The abstract performance table reveals a number of features of the CSP system. Most functions can operate at a sustained rate of around 70 MFLOPS. The notable exceptions to this are the FFT, which is able to take advantage of the dual pipeline structure of the FPPE architecture, and the thresholding, which consists of a large percentage of branch instructions and cannot make effective use of the pipeline structure. The graph execution requirement for data to pass through the GM between task execution shows in the "time in use" column, where even under the best of conditions the time in use remains under 90% in all cases but one. An important point to these results is that they indicate that the system can scale linearly with size, given a proper balance of functional elements. That means that the total throughput could possibly be about 70 MFLOPS times the number of FPPEs in the system.

4.4 APPLICATION MEASURES

Application based measures are taken from the operation of a realistic application on the system, and are used to gain information on how the system performs in practice. The CSP was specifically designed for radar processing, and the FPS-117 ground based radar processing was chosen to represent a typical processing application. One radar processing channel, based on the FPS-117 processing stream, is shown in figure 19, in the form of a data flow graph. The computations performed are labeled in the circles representing tasks. Between tasks, the arrows are labeled with the number of task iterations by the data set size for each iteration.

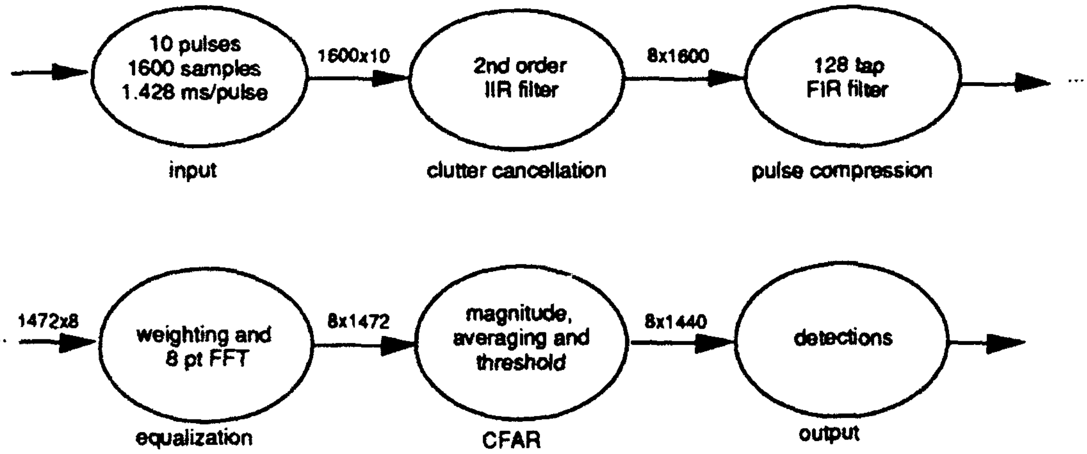


Figure 19. FPS-117 Ground Based Radar - Single Channel Data Flow Graph

The simulation of the graph execution is set up to reflect the real-time requirements of the system. The input data is a series of complex samples of the return signal from a transmitted radar pulse being sampled at a rate of 1.25 MHz. For each pulse, there are 1,600 samples of range data. Ten pulses make up one coherent integration time (CIT) interval of about 14 ms. Each datum arrives as one 32-bit word, representing the 16-bit fixed point real and imaginary parts of a complex number. There are five stages of processing to be done on the data: clutter cancellation, pulse compression, equalization, magnitude calculation, and constant false alarm rate detection. Each stage is a task that is performed on different processing elements and the stages are performed in parallel by pipelining the operations; that is, the results of one stage of processing is fed in as the data source for the next stage of processing.

Two alternative mappings of tasks to processors on the CSP were performed. The first, depicted in figure 20, computes one CIT each processing cycle and the second, depicted in figure 21, computes two CITs in each processing cycle. Time line displays of the simulation results are shown in figures 22 and 23. The FIR task, split between FPPEs 2 and 3, is shown being executed on the time line on the lines labeled PE 2 and PE 3. The magnitude and thresholding tasks are combined on PE 4. The time line on the single cycle mapping shows that FPPEs 2 and 3 are not fully utilized. The alternative double cycle mapping execution shown in figure 23 executes the FIR task for one full CIT on one FPPE. This saves the overhead of one GM access and frees up enough time in which to compute the magnitude task on the same FPPE. This results in a higher utilization of FPPEs 2 and 3, and additional available time for processing at FPPE 4. This time is not used, but is spare processing capacity available for unforeseen requirements that was not available on the single cycle mapping.

An important feature to note on the SI Out line of figure 23 is that the output latency has been doubled. This results from an inflexibility in the task scheduling of the graph execution. Although data is available for one task to start on FPPE 2 before FPPE 3, there is no control mechanism for executing one before the other, because the data for each comes from the same source.

The computation speed measurement summaries for the two mappings are presented in tables 6 and 7, using the same format as table 5. The difference between the two tables, in the Time in Use column, reflects the difference in utilization for the FIR task that is discussed above. In accordance with our three level measurement benchmark method, comparison of these tables with table 5, which reflects the abstract measures, reveals some characteristics of the system performance. The operating efficiency of the IIR task is 21.5%, far below the 50% figure of the abstract measure. This points out the inefficiency of using library macros that are generalized to a range of computation parameters. In this case, the IIR being performed is second order, while the macro is written to perform as much as a fourth order IIR operation. To improve the operating efficiency of this task, a specialized macro would be needed. In another point of interest, the FIR task registers 54% to 57% operating efficiency, which is higher than the abstract measure of the complex FIR task. This was obtained by altering the FIR algorithm to use FFT macros to perform the convolution in the frequency domain, and taking advantage of the high efficiency with which the FPPE performs the FFT. On the other hand, the low operating efficiency of the CFAR task is expected, because of the contribution from the threshold processing, which measured very low efficiency as an abstract measure. In addition to these observations, the time in use measures show that obtaining the abstract levels of usage of the FPPE is difficult, and that 75% is in practice a reasonable expectation.

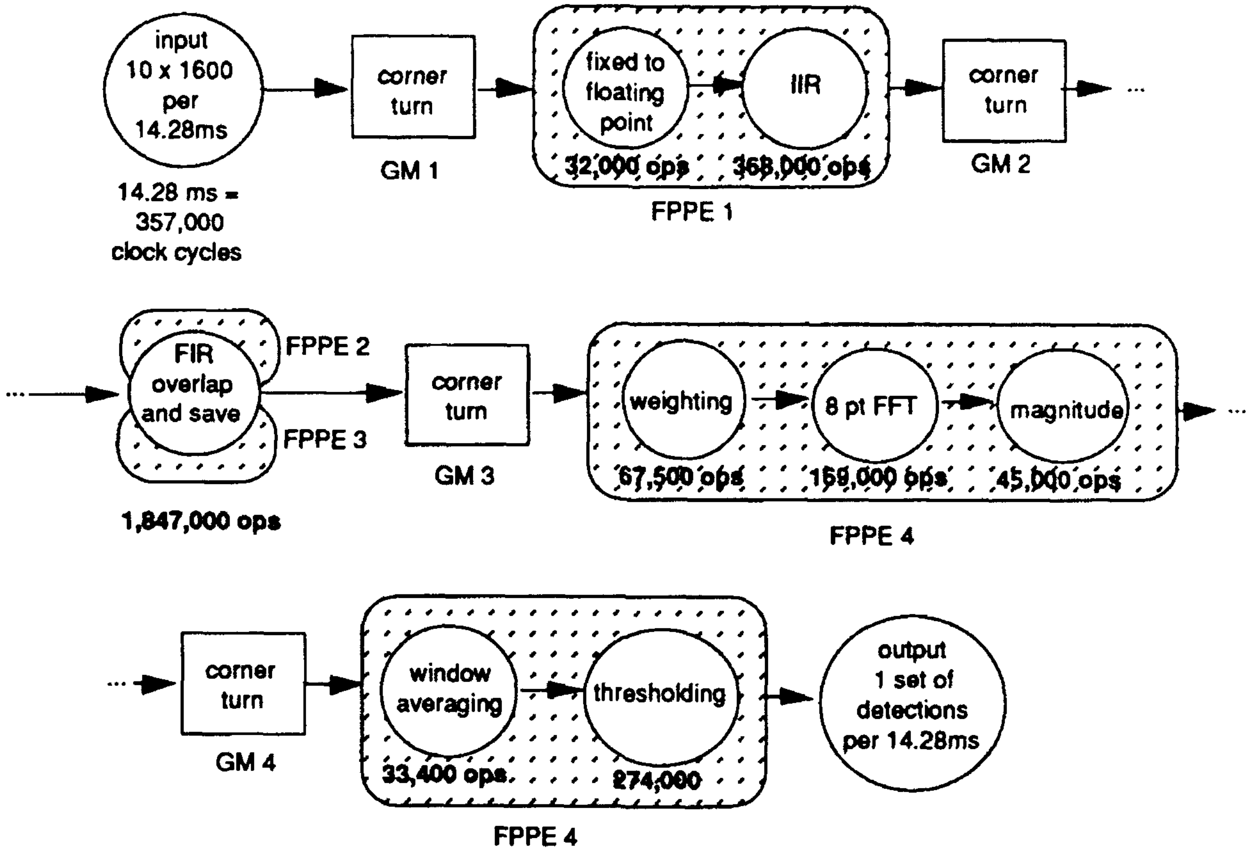


Figure 20. Single CIT/Cycle Mapping of 117 Processing to CSP 4 FPPE Configuration

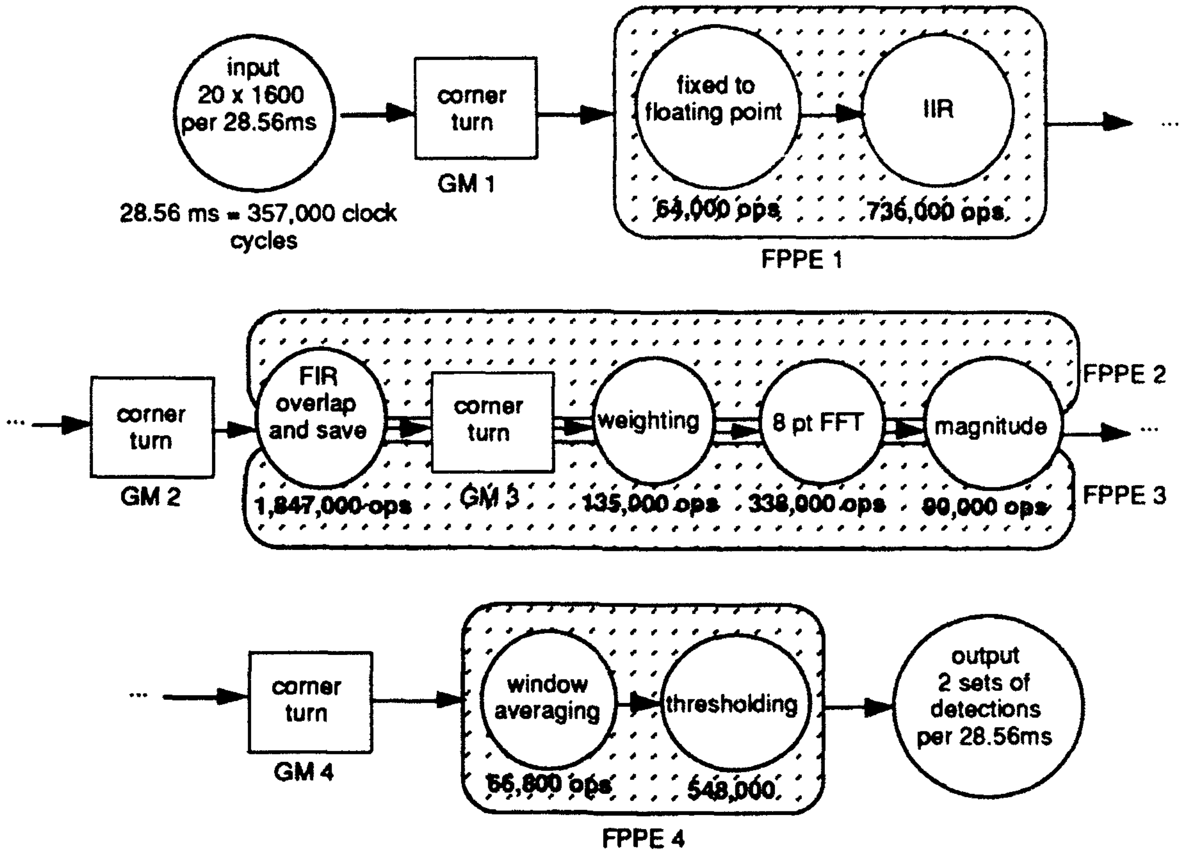


Figure 21. Double CIT/Cycle Mapping of 117 Processing to CSP 4 FPPE Configuration

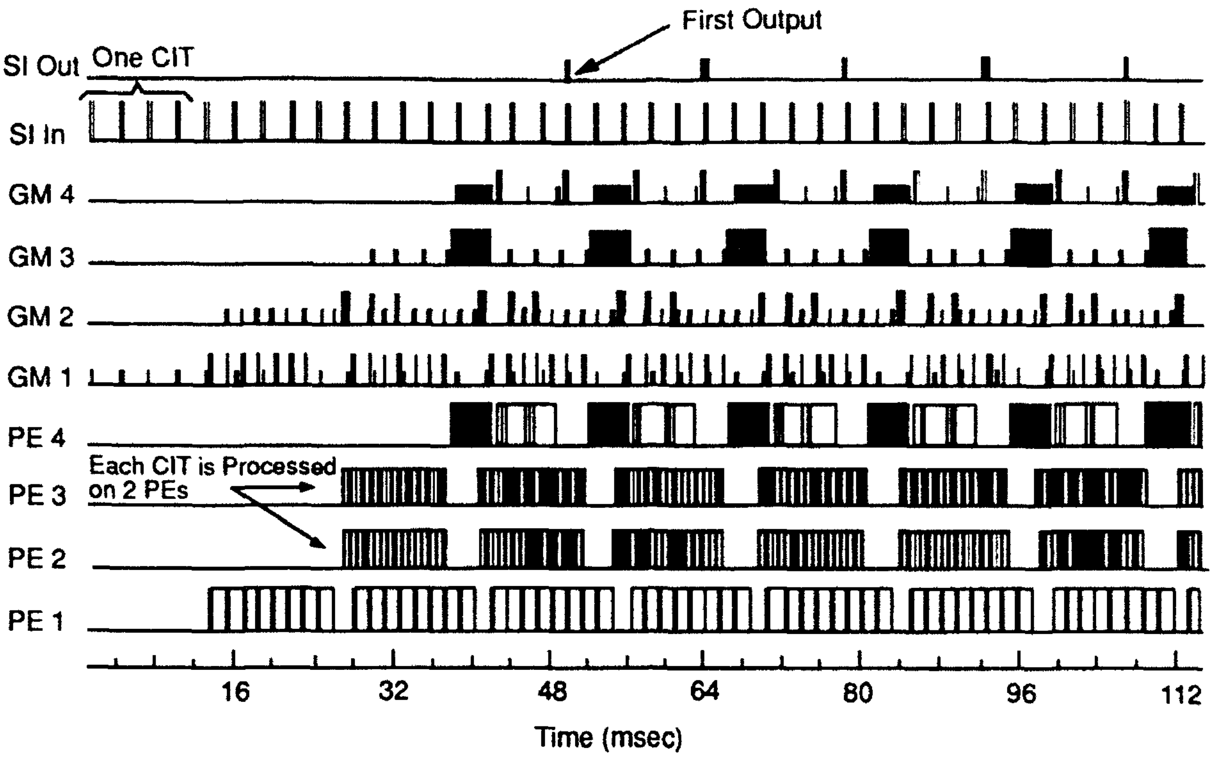


Figure 22. FPS-117 Radar Application Single CIT/Cycle Mapping

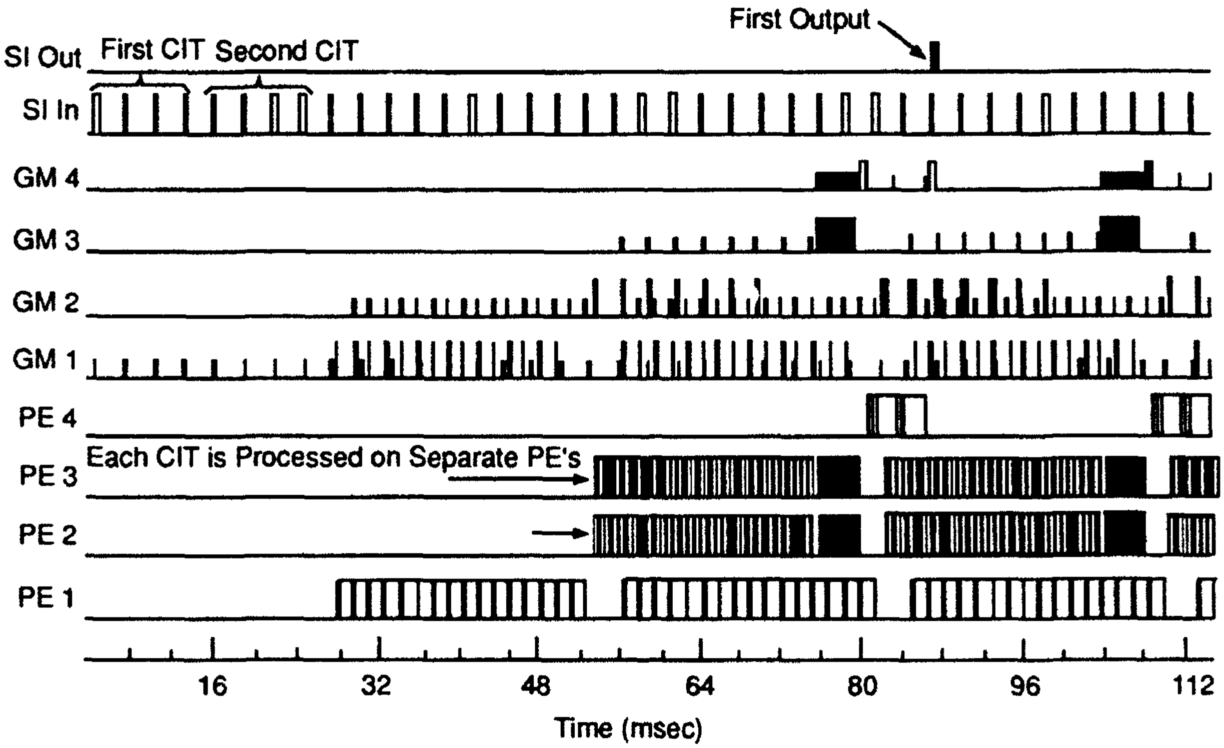


Figure 23. FPS-117 Radar Application Double CIT/Cycle Mapping

Table 6. Processing Element Application Based Performance
Single Cycle Mapping

Task	Number of Operations	Total Throughput MFLOPS	Time in Use	Operating Speed MFLOPS	Operating Efficiency
IIR	400,000	28	86%	32.5	22%
FIR (2 processors)	1,847,296	129.2	75.8%	170.4	57%
Equalization & CFAR	588,900	41	75.7%	54.4	36%
Total	2,836,196	198.2	78.3%	257.3	43%

Table 7. Processing Element Application Based Performance
Double Cycle Mapping

Task	Number of Operations	Total Throughput MFLOPS	Time in Use	Operating Speed MFLOPS	Operating Efficiency
IIR	800,000	28	86%	32.5	21.5%
FIR Equalization (each processor)	2,128,796	74.4	91%	81.8	54%
CFAR	614,800	21.2	42 .8%	50	34.8%
Total	5,672,396	198.3	78%	256	42.6%

4.5 SOFTWARE DEVELOPMENT TIMES

Entering a data flow graph specification to the CSP Graph Translator requires a detailed description of all data transfers, buffer accesses, and data type specifications. Data accesses on the GM must be exactly coordinated with the corresponding access on the FPPE. This is done with a list of commands for each functional element, with each command requiring five or six arguments. The compilation of the graph is a two step process. The first stage checks for syntactic and semantic errors. The second stage checks for inconsistencies in the specification of data transfers, such as the match between data sets and buffer sizes. The compilation time is quite long, as shown in table 8. A user with some programming experience can cut the time required for programming a graph significantly, but the graph description process itself is inherently time consuming. Table 8 is based on my experience with the software development tools running on a time-shared MicroVax workstation. The numbers reflect an approximate average of times required for each step of the software development process.

Table 8. Software Development Times

First time application development	3 months
Second time application development	3 weeks
Approximate Graph compilation time	
- Using a time-shared MicroVax	
stage 1	20 minutes
stage 2	35 minutes
simulation time	45 minutes

4.6 RESULTS ANALYSIS

The process of measuring the performance of the CSP has provided two levels of information. The quantitative performance numbers presented in the previous sections provide some understanding of the system, but apply only to particular application cases. To gain a fuller understanding of the system, we draw from the experience of measuring the performance of the system. This section of the report is based on judgements arrived at during the trial and error process of developing applications on the system. Four aspects of the system architecture are discussed here: the system execution model, inter-element data transfers, processing elements, and software development.

4.6.1 System Execution Model

The CSP uses a static method of assigning tasks to FPPEs. Construction of the application data flow graph specification includes specific hardware assignments and data packet sizes. This method uses system control mechanisms that are significantly simpler than those of a dynamic assignment method. One advantage to using a static assignment constraint is that effective system simulation is possible. The CSP simulation capabili‚; is of great value in application development, especially in helping to tune system performance. The static assignment method also helps keep task scheduling overhead low during execution, but has the disadvantage of limiting system flexibility. It can also be a hindrance when trying to minimize system latency. 'l his is a consideration, for example, when a number of tasks are waiting on the same data source. It may be the case that sufficient data is available for one of the tasks to execute before the complete data set has arrived at the storage object, but since only one data threshold may be specified for each storage object, data from the source and space for the results must be available for all tasks before any are scheduled to execute. This characteristic makes the system work best for applications with coarse grained parallelism and tasks that can be evenly matched in terms of execution time.

4.6.2 Data Transfers

There are three steps involved in the transfer of data between processing elements: data is passed from the source over the Data Network, stored in a Global Memory, and then passed over the Data Network again to its destination. The Data Network has a very high instantaneous bandwidth, with numerous paths able to run simultaneously at nearly 25 MHz each. The network can also be expanded to meet system requirements as necessary. Access to Global Memories can be performed at the data network speed. In addition, the Global Memory has a powerful set of addressing modes, including comer turning, that can be used without extra overhead. Storage of data in the Global Memories is, however, a required stage between task executions. Therefore, the Data Network and the Global Memory must be considered together as a data transfer resource, and as such, present two problems. System overhead between memory accesses is far greater than the time taken to perform the data transfers, and reduces the effective data transfer rate to one-third to one-fourth of the 25 MHz at which the data network operates (refer to instantaneous data transfer and total data throughput measures in table 3). The other disadvantage to this use of the Global Memory is that it could become a bottleneck if it is in more than one data transfer path as we found in figure 16. Because tasks are scheduled simultaneously and there is only one access port on the Global Memory element, contention for data at the Global Memory can delay task scheduling. Care must be taken in mapping the application to ensure that there are enough Global Memory elements in the system configuration to avoid such contention.

4.6.3 Processing Elements (FPPEs)

Two features of the Floating Point Processing Element add significantly to its high level of performance. A function generator allows it to perform division, square root, and exponential functions in one or two clock cycles. This ability lends more strength to the MFLOPS rating of the device, because these functions often take 30 floating-point operations or more to be computed on other processors.

Another strength of the FPPE is its memory structure. Data memory is double buffered, to allow the processor to operate on one data set while I/O operations are being performed on another data set. This feature allows the processor to attain a very high operating efficiency. Programming experience showed, however, that the small data memory size (16,384 words) limits task sizes. The maximum size complex FFT that the FPPE can perform, for example, is 4,096 points (the FFT macros did not use an "in place" algorithm). In addition, because some tasks perform more operations per data point than others, this may limit the user's ability to balance task sizes and gain maximum FPPE utilization.

Another cost of the complex nature of the FPPE structure is the complexity of programming the device. The device is programmed in a proprietary microcode that requires considerable expertise to use. While the macro library is comprehensive enough to provide for most radar signal processing tasks, a major investment is required for optimizing new functions by writing new macros or modifying existing macros.

4.6.4 Software Development

The set of software development tools included in the CSP system is comprehensive. A well structured programming methodology is supported by a clear recognition implicit in the tool set that the system engineering is performed on distinct levels. By creating a clear division and structured interfaces between the system level, data flow graph execution level, and processing element level, the tools support a team approach to application design and enhance productivity and maintainability of application software.

On the other hand, the process of data flow graph specification development is very time consuming. Each new compilation of the graph and simulation of the graph execution cycle takes about one and one-half hours on a MicroVax. This would be shorter on a more powerful platform, but is nevertheless a long recompilation cycle. The use of graph execution variables helped shorten the cycle, but use of an incremental graph recompilation method, or some other improved compiler technology, might improve turn around time. Another problem is that the graph specification language contains a lot of detailed buffer specification that could be automatically generated by a more sophisticated graph specification tool. Such a tool, perhaps with a graphical interface, would significantly enhance productivity of application developers.

4.6.5 Summary

In the final analysis, a system can only be judged against the stated objectives of the design. The system design began in 1984, at a time when technology was not available to provide 1,000 MFLOPS peak performance systems. The system was conceived to allow inclusion of future technology improvements, and in this it has succeeded. System upgrades are continuing, with future improvements currently planned to provide a factor of 4 improvement in memory and processing capacity. While programming the system is an awkward process, it is true that an elegant programming process was not a goal of the system design. The use of a static task execution model has limited the scope of applicability of the system, but at the same time allows for low processing element execution overhead and has avoided requirements for increased software development tool support for developing highly parallelized applications. However, although this system has successfully avoided the requirement for dynamic execution assignment and more comprehensive support for parallel processing software, it will likely be a requirement of future large scale parallel processing systems.

SECTION 5

CONCLUSION

The goal of this project's performance measurement work with the Common Signal Processor was to develop an approach to and gain an understanding of the problem of system performance measurement of parallel processing systems in the field of digital signal processing. In the previous section, the results of the CSP measurements are presented essentially as an example of performance measurement that follows the approach outlined in section 2. This section summarizes the method used on the CSP, comments on the value and limitations of the approach taken in relation to the goals of benchmarking outlined in the introduction, and points to further work to be done to improve the process.

The prerequisite for system measurement is availability of a system for testing. It is theoretically possible to study a system architecture and prescribe some software to run on it, in order to have it applied at a remote site by the technicians operating the system. However, a major aspect of learning what the measures mean in context comes from operating the system. This requires that the system be available for the development of the software as well as the execution of prepared applications.

The role of the quantitative measures presented in this report have been consciously deemphasized. Classically understood measures such as data network bandwidth and processing element throughput may not be clear when presented without the context. For instance, if a data transfer involves a store and forward step and becomes an integral part of the task scheduling mechanisms, it is not comparable to another system that uses a point-to-point transfer mechanism. Or, if a processing element contains support modules to perform a square root in one clock cycle, the square root operation can only be counted as one real floating point operation. This is not, however, the same floating point operation on a different system that performs the same calculation with 30 floating point operations. This feature of performance measurement is likely to become more pronounced in the future, if current trends in increased levels of integration and parallelism continue. In the absence of a common programming model, useful performance measures must be reported with significant amounts of context and subjective user experience.

The effectiveness of architectural features must be judged against their intended function. In the case of the CSP, for example, if dynamic reconfiguration was not a requirement of the original target applications when the system was designed, the feature which gains high processing element utilization at the expense of dynamic configuration is a successful feature. The scheduling overhead time should be judged, however, in comparison to other systems that also use a static task assignment execution model.

In the work presented here, a structure is provided with a three step approach outlined in section 2. The first step is to study the physical aspects of the system and its primary functional elements. This represents a preliminary description of the theoretical system capability, and provides the basic understanding of the system required to proceed with the measurement process. The next two steps, testing an application and taking abstract measures, are mutually supportive aspects of the measurement process. Programming the application tests the software development tools and leads to an understanding of system operation. The performance measurements it yields are, however, specific to one application only. The second set of measures, the abstract measures, can be made with validity only after some familiarity with the system has been obtained, and are therefore made after the application measures.

As stated in the introduction, comparing different digital signal processing systems with a "benchmark" is hindered by the absence of a common programming model. The abstract measures provide the "benchmark" against which to measure a single system performance. It serves to stress the system, thereby showing its limitations. A comparison of the abstract measures against the peak measures usually will demonstrate the system bottlenecks, and actual performance limits and shows how effective the system control mechanisms are in utilizing the system resources. It also lends some reality to the peak measures that are always the most general means of describing system performance. On the other hand, abstract measures compared to the application measures indicate how close to optimal the application performance is and how well the system can be utilized for the class of problems represented by the application. The comparison may also indicate potential for improvement in application performance and advantages and disadvantages to particular architecture features.

There is a basic issue not addressed by the approach used here. In the absence of a programming model common to the class of systems that we are targeting, the capacity to perform specific applications can be effectively tested in an absolute sense only by applying operations to the system that are similar to the application. Doing this, however, is not necessarily the process of benchmarking. In particular, it does not save the programmer the trouble required to program his application before he can learn about the system's ability to perform it.

In the continuation of this work, plans for the coming year include repeating the same process on digital signal processors with architectures different from the CSP. In the process of examining other systems, the performance measurement approach will be further refined. However, one result of the work thus far is a subjective judgment that a major improvement in the generality of the benchmarking process requires a fundamental change in the process of software development.

There are elements of the benchmark effort that can be used toward developing a new way of developing software. As a conclusion to this report on "work in progress," what follows is a brief outline of our current concept of how the process of software development can be redefined. The concept is analogous to the progression of accepted program development practice on general purpose uniprocessor systems from the use of assembly language to the wide spread use of high level languages. Languages such as C and Fortran are now used with little knowledge needed of the system to which they are being applied. They provide the programmer with operation constructs such as iterations and branching structures and data constructs such as arrays and pointers that capture the fundamental building blocks of the software while hiding the underlying details of the system architecture. The programmer uses the high level language to express the processor instructions and the compiler translates the program to machine language.

In the same way, we want to find the set of operations and data structures that are fundamental to describing digital signal processing applications, independent of the architecture upon which they are to be executed. Example data structures are matrices and trees and example operations are matrix multiplication and inversion or searches and sorts of elements in a tree.

Efforts to find a fundamental set of operations that apply performance measurement to a variety of systems naturally extend to finding the elements of applications common across different systems. They also serve the same objectives of the benchmarking effort, that is to improve understanding of existing systems and the architecture features required for fundamental improvements in performance. Just as Reduced Instruction Set Computer architectures have developed to support efficient compilation of high order languages on a general purpose uniprocessor system, it is natural to expect the acceptance of digital signal processing instruction set powerful enough to gain acceptance across different systems will eventually lead to the development of architectures that are intended to support its efficient implementation. Regardless of the compilation process that applies instructions to the system, one set of operations could be applied to a number of systems to make a valid comparison of performance.

LIST OF REFERENCES

1. Nowacki, C. L., and L. A. Crook, October 1991, *Design of the Open Modeling System*, MTR-11261, The MITRE Corporation, Bedford, MA.

2. Cenkl, M., and C. L. Nowacki, October 1991, *Techniques for Mapping Algorithms to Parallel Signal Processors,* MTR-11266, The MITRE Corporation, Bedford, MA.

3. Michaud, M. C., G. M. Whitaker, J. B. Goethert, and M. A. Schroeder, October 1991, *Parallelism in Signal Processing Algorithms: Final Interim Report*, MTR-10824, The MITRE Corporation, Bedford, MA.

4. Lyon, Gordon, 1989, *Design Factors for Parallel Processing Benchmarks*, Theoretical Computer Science, North-Holland, Amsterdam, Vol. 64, pp. 175-189.

5. Ferrari, D. G. Serazzi, and A. Zeigner, 1983, *Measurement and Tuning of Computer Systems,* New Jersey: Prentice-Hall, Inc.

6. Jain, R., 1991, *The Art of Computer Systems Performance Analysis,* New York: John Wiley & Sons.

7. *Common Signal Processor Applications User's Guide*, Document Number 118A636, 7 September 1990, International Business Machines Technical Document.

☆U.S. GOVERNMENT PRINTING OFFICE· 1993-710-093-60227

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.